Chapel: Parallel Programming Made Productive

Brad Chamberlain
Cray Inc.

Seattle University — May 2nd, 2012
What is Parallel Computing?

Parallel Computing:
What is Parallel Computing?

Parallel Computing: Using multiple processors and their memories to execute a computation.

Why would we do this?
Parallel Computing: Using multiple processors and their memories to execute a computation.

Why would we do this?

• To execute a program more quickly than you otherwise could
• Or, potentially, to execute a larger program (in terms of data) than you otherwise could
Parallel Computations Vary in Difficulty
Matrix Addition: Quite straightforward
Matrix Multiplication: Far more involved
Some Terminology

**processor core (or simply “core”):** the unit of a computer that has a PC, executes instructions, etc.

**node:** a group of cores and memories that must go over the network to communicate with any others

**network:** the wires and chips that permit nodes to communicate with one another
Jaguar

- Most Powerful Supercomputer in the US
- nodes: 18,688
- memory: 32GB/node = 584 TB total
- procs: 16-core AMD/node = 299,008 cores
- GPUs: 960
- peak speed: 3.3 petaflops
- floorspace: 4,352 square feet
Top500: Simply One Piece of the Puzzle

- Rates the 500 fastest computers twice a year
- Measured using the LINPACK benchmark
  - Solves an LU factorization
  - Flops dominate runtime
- Yet, other factors limit most real applications
  - e.g., memory bandwidth

### TOP500 List - November 2011 (1-100)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Computer/Year Vendor</th>
<th>Cores</th>
<th>R_{max}</th>
<th>R_{peak}</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KEIKEN Advanced Institute for Computational Science (AICS) Japan</td>
<td>K computer, SPARC64 VIIIfx 2.6GHz, Tohuku Interconnect 2011 Fujitsu</td>
<td>705624</td>
<td>10510.00</td>
<td>11280.38</td>
<td>12559.9</td>
</tr>
<tr>
<td>2</td>
<td>National Supercomputing Center in Tanya China</td>
<td>NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2090/2010 NUDT</td>
<td>186398</td>
<td>2560.00</td>
<td>4780.00</td>
<td>4090.0</td>
</tr>
<tr>
<td>3</td>
<td>DOE/SC/Oak Ridge National Laboratory United States</td>
<td>Cray XT5+ HE Opteron 6-core 2.6 GHz J 2009 Cray Inc.</td>
<td>224160</td>
<td>1759.00</td>
<td>2331.00</td>
<td>6950.0</td>
</tr>
<tr>
<td>4</td>
<td>National Supercomputing Center in Shenzen (NSC5) China</td>
<td>Dawning TC3500 Blade System, Xeon X5550 6C 2.66GHz, Infiniti QDR, NVIDIA 2050/2010 Dawning</td>
<td>120640</td>
<td>1271.00</td>
<td>2964.36</td>
<td>2586.0</td>
</tr>
<tr>
<td>5</td>
<td>GSIC Center, Tokyo Institute of Technology Japan</td>
<td>HP Proliant SL350s G7 Xeon 6C X5679, Nvidia GPU, Linux/Windows 2010 NSC5H</td>
<td>73278</td>
<td>1192.00</td>
<td>2287.53</td>
<td>1398.6</td>
</tr>
<tr>
<td>6</td>
<td>DOE/SC/NLANR/BNL United States</td>
<td>Cray XE6, Opteron 6136 8C 2.4GHz Custom / 2011 Cray Inc.</td>
<td>142272</td>
<td>1110.00</td>
<td>1369.31</td>
<td>3980.0</td>
</tr>
<tr>
<td>7</td>
<td>NASA Ames Research Center/NAS United States</td>
<td>SG Altix ICE 5350EX/400EX Xeon HT QC 3.33Ghz, 56706702.6GHz, Intel/2011 G5</td>
<td>111104</td>
<td>1688.00</td>
<td>1315.33</td>
<td>4102.0</td>
</tr>
<tr>
<td>8</td>
<td>DOE/SC/NLANR/RSC United States</td>
<td>Cray XE6, Opteron 6172 12C 2.1GHz Custom / 2010 Cray Inc.</td>
<td>153408</td>
<td>1054.00</td>
<td>1288.63</td>
<td>2910.0</td>
</tr>
<tr>
<td>9</td>
<td>Commissionat de l'Energie Atomique (CEA) France</td>
<td>Bull Bull super-node 58001565030 J 2010 Bull</td>
<td>136386</td>
<td>1650.00</td>
<td>1254.56</td>
<td>4590.0</td>
</tr>
<tr>
<td>10</td>
<td>DOE/SC/NLANR/BNL United States</td>
<td>BladeCenter QS22, L21 Cluser, PowerXCell 8i 3.2 GHz / Opteron DC 1.8 GHz, Vertano Infiniti J 2009 IBM</td>
<td>122400</td>
<td>1642.00</td>
<td>1375.78</td>
<td>2345.0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Date</th>
<th>System</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GF</td>
<td>1988</td>
<td>Cray Y-MP</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Static finite element analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fortran77 + Cray autotasking + vectorization</td>
<td></td>
</tr>
<tr>
<td>1 TF</td>
<td>1998</td>
<td>Cray T3E</td>
<td>1,024</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modeling of metallic magnet atoms</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fortran + MPI (Message Passing Interface)</td>
<td></td>
</tr>
<tr>
<td>1 PF</td>
<td>2008</td>
<td>Cray XT5</td>
<td>150,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Superconductive materials</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C++/Fortran + MPI + vectorization</td>
<td></td>
</tr>
<tr>
<td>1 EF</td>
<td>~2018</td>
<td>Cray</td>
<td>~10,000,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TBD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TBD: C/C++/Fortran + MPI + CUDA/OpenCL/OpenMP/OpenACC</td>
<td></td>
</tr>
</tbody>
</table>

Or Perhaps Something Completely Different?
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:**

\[
\begin{align*}
A & \quad = \\
B & \quad + \\
C & \quad \cdot \\
\alpha & 
\end{align*}
\]
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):**
```
#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;
}
```
#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;
}
```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];
}
```

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

Examples:

<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
A Possible Reaction:

“This is all well and good for HPC users, but I’m a mainstream desktop programmer, so this is all academic for me.”

The Unfortunate Reality:

• Performance-minded mainstream programmers are forced to deal increasingly with parallelism too (due to multicore)
• And, as chips become more complex, locality too
Next-generation HPC Processor Technologies

Intel MIC

AMD Fusion

Nvidia Echelon

Tilera Tile-Gx

Sources:
http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg
http://www.thinkcomputers.org/ces-2011-amds-fusion-apus/
http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf
Increased hierarchy and/or sensitivity to locality
Heterogeneous processor and memory types

⇒ Both HPC and mainstream programmers will have a lot more to think about at the processor level
Rewinding a few slides...

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

```
#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;
}
```

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

#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;
  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;
}

#ifdef _OPENMP
#pragma omp config config m = 1000,
    alpha = 3.0;
#endif

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.
Outline

✓ Motivation

➢ Chapel Background and Themes
  • Tour of Chapel Concepts
  • Project Status
What is Chapel?

- An emerging parallel programming language
  - Design and development led by Cray Inc.
    - in collaboration with academia, labs, industry
  - Initiated under the DARPA HPCS program

- **Overall goal:** Improve programmer productivity
  - Improve the **programmability** of parallel computers
  - Match or beat the **performance** of current programming models
  - Support better **portability** than current programming models
  - Improve the **robustness** of parallel codes

- A work-in-progress
Chapel's Implementation

• Being developed as open source at SourceForge

• Licensed as BSD software

• **Target Architectures:**
  • Cray architectures
  • multicore desktops and laptops
  • commodity clusters
  • systems from other vendors
  • *in-progress:* CPU+accelerator hybrids, manycore, ...
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 all parallelism available in the hardware

- **Types**: machines, nodes, cores, instructions

<table>
<thead>
<tr>
<th>Style 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>
In pictures: “Apply a 3-Point Stencil to a vector”

\[
\text{Global-View} \quad \frac{1}{2} \sum (\text{vector}) + \text{vector} = \text{vector}
\]

\[
\text{Local-View}
\]
2) Global-View Abstractions

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

**Global-View**

\[
\begin{align*}
& (\quad \quad \quad )/2 \\
+ & (\quad \quad \quad )/2 \\
= & (\quad \quad \quad )/2
\end{align*}
\]

**Local-View**

\[
\begin{align*}
& (\quad \quad \quad )/2 \\
+ & (\quad \quad \quad )/2 \\
= & (\quad \quad \quad )/2
\end{align*}
\]

\[
\begin{align*}
& (\quad \quad \quad )/2 \\
+ & (\quad \quad \quad )/2 \\
= & (\quad \quad \quad )/2
\end{align*}
\]
2) Global-View Abstractions

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

**Global-View**

```plaintext
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)**

```plaintext
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”

```
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;
}
```

```
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;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Assumes p divides n
2) rprj3 Stencil from NAS MG

\[ w_0 = w_1 = w_2 = w_3 \]
subroutine norm2u3(r,n1,n2,n3,rnm2,rnmu,nx,ny,nz)

implied none

double precision rnm2, rnmu
integer n1, n2, n3,
include 'globals.h'

implicit none

subroutine rnm2=
  s =
endif
enddo

else
if(m3k.eq.3)then
endif
else
if(m1k.eq.3)then
endif
endif

subroutine rprj3( r,m1k,m2k,m3k,s,m1j,m2j,m3j,k )
d3 = 2
d2 = 1
endif
denddo
d2 = 1
diff(1,1,1)then
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endi
2) rprj3 Stencil from NAS MG in Chapel

```chapel
proc rprj3(S: [?SD], R: [?RD]) {
    const Stencil = [-1..1, -1..1, -1..1],
    W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
    W3D = [(i,j,k) in Stencil] W[(i!=0) + (j!=0) + (k!=0)];

    forall ijk in SD do
        S[ijk] = + reduce [offset in Stencil]
            (W3D[offset] * R[ijk + RD.stride*offset]);
}
```

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

<table>
<thead>
<tr>
<th>Communication Libraries</th>
<th>System</th>
<th>Data Model</th>
<th>Control Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPI/MPI-2</td>
<td>Local-View</td>
<td>Local-View</td>
</tr>
<tr>
<td>SHMEM, ARMCI, GASNet</td>
<td></td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>OpenMP, Pthreads</td>
<td>Global-View (trivially)</td>
<td>Global-View (trivially)</td>
</tr>
<tr>
<td>PGAS Languages</td>
<td>Co-Array Fortran</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td></td>
<td>UPC</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td></td>
<td>Titanium</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>PGAS Libraries</td>
<td>Global Arrays</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>HPCS Languages</td>
<td>Chapel</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td></td>
<td>X10 (IBM)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td></td>
<td>Fortress (Sun)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
</tbody>
</table>
A language may support both global- and local-view programming — in particular, Chapel does.

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

proc MySPMDProgram(me, p) {
  ...
}
```
3) Multiresolution Design: Motivation

“Why is everything so tedious/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 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 computation affects performance and scalability
- Give programmers control of data and task placement
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 alienating the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional
Outline

✓ Motivation
✓ Chapel Background and Themes
➢ Tour of Chapel Concepts
• Project Status
Base Language Features

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
const pi = 3.14, // pi is a real
    coord = 1.2 + 3.4i, // coord is a complex...
    coord2 = pi*coord, // ...as is coord2
    name = "brad", // name is a string
    verbose = false; // verbose is boolean

proc addem(x, y) { // addem() has generic arguments
    return x + y; // and an inferred return type
}

var sum = addem(1, pi), // sum is a real
    fullname = addem(name, "ford"); // fullname is a string

writeln((sum, fullname));

(4.14, bradford)
const r = 1..10;

printVals(r # 3);
printVals(r # -3);
printVals(r by 2);
printVals(r by 2 align 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);

proc printVals(r) {
    for i in r do
        write(r, " ");
        writeln();
}
Iterators

iter fibonacci(n) {
    var current = 0,
        next = 1;
    for 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}

for f in fibonacci(7) do writeln(f);

0 1 1 2 3 5 8

iter tiledRMO(D, tilesize) {
    const tile = [0..#tilesize, 0..#tilesize];
    for base in D by tilesize do
        for ij in D[tile + base] do
            yield ij;
}

for ij in tiledRMO(D, 2) do write(ij);

(1,1) (1,2) (2,1) (2,2)
(1,3) (1,4) (2,3) (2,4)
(1,5) (1,6) (2,5) (2,6)
...
(3,1) (3,2) (4,1) (4,2)
for (i,f) in (0..#n, fibonacci(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
Other Base Language Features

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

...
Task Parallel Features

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

```coforall
coforall t in 0..#numTasks do
  writeln("Hello from task ", t, " of ", numTasks);
  writeln("All tasks done");
```

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
cobegin {
    producer();
    consumer();
}

// 'sync' types store full/empty state along with value
var buff$: [0..#buffersize] sync real;

proc producer() {
    var i = 0;
    for ... {
        i = (i+1) % buffersize;
        buff$[i] = ...;  // reads block until empty, leave full
    }
}

proc consumer() {
    var i = 0;
    while ... {
        i = (i+1) % buffersize;
        ...buff$[i]...;  // writes block until full, leave empty
    }
}
Other Task Parallel Features

- *begin* statements for fire-and-forget tasks
- *atomic variables* for lock-free programming
Locality Features

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
Definition:
- Abstract unit of target architecture
- Supports reasoning about locality
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Typically: A multi-core processor or SMP* node
Defining 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 = ...;
  ```

  Locales:  L0  L1  L2  L3  L4  L5  L6  L7
Locale Operations

• Locale methods support queries about target system:

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

• On-clauses support placement of computations:

```plaintext
writeln("on locale 0");
on Locales[1] do
  writeln("now on locale 1");
writeln("on locale 0 again");

cobegin {
  on A[i,j] do
    bigComputation(A);
  on node.left do
    search(node.left);
}
```
Data Parallel Features

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

Chapel supports several types of domains (index sets):

- **dense**
- **strided**
- **sparse**
- **associative**
- **unstructured**
Chapel Array Types

Each domain type can be used to declare arrays:

- **dense**
- **strided**
- **sparse**
- **associative**
- **unstructured**
Data Parallel Operations

- Parallel Iteration via `forall` loops
  
  \[ A = \text{forall} (i, j) \text{ in } D \text{ do } (i + j/10.0); \]

- Array Slicing; Domain Algebra
  
  \[ A[\text{InnerD}] = B[\text{InnerD} + (0,1)]; \]

- Promotion of Scalar Operators and Functions
  
  \[ A = B + \alpha \times C; \quad A = \exp(B, C); \]

- And several others: indexing, reallocation, set operations, remapping, aliasing, queries, ...
Jacobi Iteration in Pictures

\[ \sum \left( \begin{array}{c}
\text{1.0}
\end{array} \right) \div 4 \]

repeat until max change < \( \varepsilon \)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A[i-1,j] + A[i+1,j]
        + A[i,j-1] + A[i,j+1]) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;
    const delta = max reduce abs(A(D) - Temp(D));
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Declare program parameters

- `config` ⇒ can be set on executable command-line
  ```prompt> jacobi --n=10000 --epsilon=0.0001```
- `const` ⇒ can’t change values after initialization
- `var` A, Temp : [BigD] real
- `note` that no types are given; inferred from initializer
  ```prompt>```
- `n` ⇒ default integer (32 bits)
- `epsilon` ⇒ default real floating-point (64 bits)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

for i in D do
    for j in D
        Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4;

var delta = max reduce abs(A(D) - Temp(D));
A(D) = Temp(D);
```

Declare domains (first class index sets)

domain(2) $\Rightarrow$ 2D arithmetic domain, indices are integer 2-tuples

subdomain($P$) $\Rightarrow$ a domain of the same type as $P$ whose indices are guaranteed to be a subset of $P$'s

exterior $\Rightarrow$ one of several built-in domain generators
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
```

**Declare arrays**

- `var` can be modified throughout its lifetime
- `[BigD] T` is an array of size `BigD` with elements of type `T`
- *(no initializer)* values initialized to default value (0.0 for reals)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
D: subdomain(BigD) = [1..n, 1..n],
LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
```

**Set Explicit Boundary Condition**

- indexing by domain ⇒ slicing mechanism
- array expressions ⇒ parallel evaluation

```
```

Set Explicit Boundary Condition

- indexing by domain ⇒ slicing mechanism
- array expressions ⇒ parallel evaluation
Jacobi Iteration in Chapel

config const n = 6, epsilon = 1.0e-5;

Compute 5-point stencil

\[(i,j) \text{ in } D \Rightarrow \text{parallel forall expression over } D' \text{ indices, binding them to new variables } i \text{ and } j\]

\[
\sum \left( \begin{array}{c}
\text{+} \\
\text{+} \\
\text{+} \\
\text{+} \\
\end{array} \right) \div 4
\]

\[
\]

const delta = max reduce abs(A[D] - Temp[D]);
A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],

Compute maximum change

op reduce ⇒ collapse aggregate expression to scalar using op

Promotion: abs() and – are scalar operators, automatically promoted to work with array operands

do {
    [(i,j) in D] Temp(i,j) = (A[i-1,j] + A[i+1,j]
                            + A[i,j-1] + A[i,j+1]) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
cfg const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
    [(i,j) in D] Temp(i,j) = (A[i-1,j] + A[i+1,j]
        + A[i,j-1] + A[i,j+1]) / 4;
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard `do...while` loop construct
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A[i-1,j] + A[i+1,j]
        + A[i,j-1] + A[i,j+1]) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Write array to console
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = [0..n+1, 0..n+1] dmapped Block(...),
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

var delta = max reduce abs(A(D) - Temp(D));

while (delta > epsilon) {
    for (i,j) in D
        Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;
    A = Temp;
}
```

With this change, same code runs in a distributed manner.

Domain distribution maps indices to `locales`
- decomposition of arrays & default mapping of iterations to locales
- Subdomains inherit parent domain's distribution
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = [0..n+1, 0..n+1] dmapped Block(...),
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A[i-1,j] + A[i+1,j]
        + A[i,j-1] + A[i,j+1]) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Domain Maps

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

Domain maps are “recipes” that instruct the compiler how to map the global view of a computation...

...to the target locales’ memory and processors:

\[
A = B + \alpha \cdot C;
\]
STREAM Triad: Chapel (multicore)

```chapel
const ProblemSpace = [1..m];

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

A = B + alpha * C;
```
STREAM Triad: Chapel (multicore)

```chapel
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 indices and values
- computation will execute using local processors only
const ProblemSpace = [1..m]

dmapped Block(boundingBox=[1..m]);

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

A = B + alpha * C;
STREAM Triad: Chapel (multilocale, cyclic)

```
const ProblemSpace = [1..m]
dmapped Cyclic(startIdx=1);

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

A = B + alpha * C;
```
All Chapel domain types support domain maps.
1. Chapel provides a library of standard domain maps
   • to support common array implementations effortlessly

2. Advanced users can write their own domain maps in Chapel
   • to cope with shortcomings in our standard library

3. Chapel’s standard domain maps are written using the same end-user framework
   • to avoid a performance cliff between “built-in” and user-defined cases
For More Information on Domain Maps

**HotPAR’10:** *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*  
Chamberlain, Deitz, Iten, Choi; June 2010

**CUG 2011:** *Authoring User-Defined Domain Maps in Chapel*  
Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

**PGAS 2011:** *User-Defined Parallel Zippered Iterators in Chapel,*  
Chamberlain, Choi, Deitz, Navarro; October 2011

**Chapel release:**  
- Technical notes detailing domain map interface for programmers:  
  - `$CHPL_HOME/doc/technotes/README.dsi`  
- Current domain maps:  
  - `$CHPL_HOME/modules/dists/*.*chpl`  
  - `layouts/*.*chpl`  
  - `internal/Default/*.*chpl`
Chapel avoids locking crucial implementation decisions into the language specification
  • local and distributed array implementations
  • parallel loop implementations

Instead, these can be...
  ...specified in the language by an advanced user
  ...swapped in and out with minimal code changes

The result separates the roles of domain scientist, parallel programmer, and implementation cleanly
Motivation
Chapel Background and Themes
Tour of Chapel Concepts
Project Status
In a nutshell:

- Most features work at a functional level
- Many performance optimizations remain
  - particularly for distributed memory (multi-locale) execution

This is a good time to:

- Try out the language and compiler
- Use Chapel for non-performance-critical projects
- Give us feedback to improve Chapel
- Use Chapel for parallel programming education
If I were teaching parallel programming, I’d want to cover:

- data parallelism
- task parallelism
- concurrency
- synchronization
- locality/affinity
- deadlock, livelock, and other pitfalls
- performance tuning
- memory consistency models

I don’t think there’s been a good language out there...

- for teaching all of these things
- for teaching some of these things well at all

until now: We believe Chapel can potentially play a crucial role here

(see http://chapel.cray.com/education.html for more information)
Join Our Growing Community

- **Cray:**
  - Brad Chamberlain
  - Sung-Eun Choi
  - Greg Titus
  - Vass Litvinov
  - Tom Hildebrandt
  - (open positions)

- **External Collaborators:**
  - Albert Sidelnik (UIUC)
  - Jonathan Turner (CU Boulder)
  - Kyle Wheeler (Sandia)

- **Interns:**
  - Jonathan Claridge (UW)
  - Hannah Hemmaplardh (UW)
  - Andy Stone (Colorado State)
  - Jim Dinan (OSU)
  - Rob Bocchino (UIUC)
  - Mackale Joyner (Rice)
Featured Collaborations (see chapel.cray.com/collaborations.html for details)

- **CPU-GPU Computing**: UIUC (David Padua, Albert Sidelnik, Maria Garzarán)
  - paper to appear at IPDPS 2012
- **Tasking using Qthreads**: Sandia (Rich Murphy, Kyle Wheeler, Dylan Stark)
  - paper at CUG, May 2011
- **Interoperability using Babel/BRAID**: LLNL (Tom Epperly, Adrian Prantl, et al.)
  - paper at PGAS, Oct 2011
- **Dynamic Iterators**:
- **Bulk-Copy Opt**: U Malaga (Rafael Asenjo, Maria Angeles Navarro, et al.)
- **Parallel File I/O**:
  - paper at ParCo, Aug 2011
- **Improved I/O & Data Channels**: LTS (Michael Ferguson)
- **Interfaces/Generics/OOP**: CU Boulder (Jeremy Siek, Jonathan Turner)
- **Tasking over Nanos++**: BSC/UPC (Alex Duran)
- **Tuning/Portability/Enhancements**: ORNL (Matt Baker, Jeff Kuehn, Steve Poole)
- **Chapel-MPI Compatibility**: Argonne (Rusty Lusk, Pavan Balaji, Jim Dinan, et al.)
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 HPC architectures
...and for the growing need for parallel programming in the mainstream
Chapel project page: [http://chapel.cray.com](http://chapel.cray.com)
- overview, papers, presentations, language spec, ...

Chapel SourceForge page: [https://sourceforge.net/projects/chapel/](https://sourceforge.net/projects/chapel/)
- release downloads, public mailing lists, code repository, ...

Mailing Lists:
- chapel_info@cray.com: contact the team
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: dev.-oriented discussion
- chapel-education@lists.sourceforge.net: educator-oriented discussion
- chapel-bugs@lists.sourceforge.net: public bug forum
- chapel_bugs@cray.com: private bug mailing list