High-level Programming of Large-scale Parallel Systems with Chapel

Clemens Grelck

University of Amsterdam

Programming Large-Scale Parallel Systems
High-level Parallel Programming with Chapel

Programming Large-scale Systems: State of the Art

Chapel at a Glance

Chapel: Base Language

Chapel: Data Parallelism

Chapel: Task Parallelism

Chapel: Locales

Chapel: Domain Maps
Programming Large-scale Systems

**Message passing:**

- Some (often same) program runs on each compute node.
- Program is aware of other instances/programs/nodes.
- Instances send each other messages over network.
- Easy to implement:
  - lends itself easily to library for host language approach
Programming Large-scale Systems

Message passing:

- Some (often same) program runs on each compute node.
- Program is aware of other instances/programs/nodes.
- Instances send each other messages over network.
- Easy to implement:
  lends itself easily to library for host language approach

Industry standard implementation:

- Message Passing Interface (MPI)
- Original design: 1992
- Hundreds of routines defined for C and Fortran
- Point-to-point communication: send/receive
- Structured communication: collectives
What’s wrong with MPI?

- Nothing really, but .....
Programming Large-scale Systems

What’s wrong with MPI?

- Nothing really, but ..... 
- accessibility for sub-ninja programmers?
- programming productivity?
- programming errors and debugging?
- confidence in correct implementation of algorithms?

In essence:

- MPI is a great tool for ninja programmers.
- MPI is not so great for anyone else.

Insight:

- MPI *is* the de-facto standard today.
- BUT: it's worthwhile to know alternative approaches.
Programming Large-scale Systems

What’s wrong with MPI?

- Nothing really, but .....  
- accessibility for sub-ninja programmers?  
- programming productivity?  
- programming errors and debugging?  
- confidence in correct implementation of algorithms?

In essence:

- MPI is a great tool for ninja programmers.  
- MPI is not so great for anyone else.
What’s wrong with MPI?

- Nothing really, but ..... 
- accessibility for sub-ninja programmers? 
- programming productivity? 
- programming errors and debugging? 
- confidence in correct implementation of algorithms?

In essence:

- MPI is a great tool for ninja programmers. 
- MPI is not so great for anyone else.

Insight:

- MPI *is* the de-facto standard today. 
- BUT: it’s worthwhile to know alternative approaches.
Message Passing means Local View Programming

Local view programming:
- Data structures (arrays) explicitly fragmented among nodes
- Program instances concerned with subproblem only
- Whole problem solution only exists ...
  - in the mindset of the programmer (hopefully)
  - in the intricacies of message passing logic
- Fragmented data, e.g. array slice
- Concrete local indices vs virtual global indices

Global view programming:
- Program solves the whole problem
- Concrete and conceptual data structures coincide
- Concrete and conceptual indices coincide
- Similar to programming
  - sequential systems
  - small-scale shared memory systems
Message Passing means Local View Programming

Local view programming:
- Data structures (arrays) explicitly fragmented among nodes
- Program instances concerned with subproblem only
- Whole problem solution only exists ...
  - in the mindset of the programmer (hopefully)
  - in the intricacies of message passing logic
- Fragmented data, e.g. array slice
- Concrete local indices vs virtual global indices

Global view programming:
- Program solves the whole problem
- Concrete and conceptual data structures coincide
- Concrete and conceptual indices coincide
- Similar to programming
  - sequential systems
  - small-scale shared memory systems
Message Passing means Local View Programming

**Local view programming:**
- Data structures (arrays) explicitly fragmented among nodes
- Program instances concerned with subproblem only
- Whole problem solution only exists ...
  - in the mindset of the programmer (hopefully)
  - in the intricacies of message passing logic
- Fragmented data, e.g. array slice
- Concrete local indices vs virtual global indices
- **Fundamental source of errors**

**Global view programming:**
- Program solves the whole problem
- Concrete and conceptual data structures coincide
- Concrete and conceptual indices coincide
- Similar to programming
  - sequential systems
  - small-scale shared memory systems
Global View Programming on Distributed Memory?

Virtual shared memory:

- Create software abstraction of a shared address space across multiple nodes
- Disambiguate memory address on load operation:
  - Local address: load from memory as usual
  - Remote addresses: fetch value via network

Problems:
- Remote load several orders of magnitude slower
- In large clusters almost any address is remote
- Programmer has no control over data locality
- Data locality is crucial to achieve reasonable performance
Global View Programming on Distributed Memory?

**Virtual shared memory:**
- Create software abstraction of a shared address space across multiple nodes
- Disambiguate memory address on load operation:
  - Local address: load from memory as usual
  - Remote addresses: fetch value via network

**Problems:**
- Remote load several orders of magnitude slower
- In large clusters almost any address is remote
- Programmer has no control over data locality
- Data locality is crucial to achieve reasonable performance
Partitioned Global Address Space (PGAS)

**Principal idea:**
- Global view programming
- Global address space
- Control over data placement and locality
- Idea: combine the best of two worlds

**PGAS Languages:**
- Unified Parallel C (UPC)
- Co-Array Fortran (CAF)
- Chapel
High-level Parallel Programming with Chapel

Programming Large-scale Systems: State of the Art

Chapel at a Glance

Chapel: Base Language

Chapel: Data Parallelism

Chapel: Task Parallelism

Chapel: Locales

Chapel: Domain Maps
Chapel

**Origin:**

- Designed and developed by Cray
  - Fortress by SUN Microsystems
  - X10 by IBM
  - Chapel by Cray
- Today: open source (BSD) under Cray leadership
- Still under very active development
- Growing open source community

Achievement:

- 2012 High Performance Computing Challenge Award: Most elegant language

Clemens Grelck, University of Amsterdam

High-level Parallel Programming with Chapel
Chapel

Origin:

- Designed and developed by Cray
  - Fortress by SUN Microsystems
  - X10 by IBM
  - Chapel by Cray
- Today: open source (BSD) under Cray leadership
- Still under very active development
- Growing open source community

Achievement:

- 2012 High Performance Computing Challenge Award: Most elegant language
Chapel: Design Principles

Design goals:

- Improve programmability of parallel computers
- Match or beat performance of existing programming models
- Support better portability than existing programming models
- Improve robustness of parallel code
Chapel: Design Principles

Multi-paradigm design:

▶ Data parallelism
▶ Task parallelism
▶ Nested parallelism

Multi-resolution design:

▶ vector registers
▶ multiple cores
▶ multiple processors
▶ multiple nodes
▶ many nodes (supercomputers)
▶ (accelerators as future work)

All with one set of programming abstractions !!
Chapel: Design Principles

Programming language design:

▶ Design from scratch
▶ No established host language (as CAF, UPC)
▶ No historic baggage from sequential times
▶ No semantic ambiguities and dark corners

Partitioned global address space:

▶ Global view programming
▶ Local view programming
▶ Locality control (compute on local data)
▶ Affinity control (data alignment)
High-level Parallel Programming with Chapel

Programming Large-scale Systems: State of the Art

Chapel at a Glance

Chapel: Base Language

Chapel: Data Parallelism

Chapel: Task Parallelism

Chapel: Locales

Chapel: Domain Maps
High-level Parallel Programming with Chapel

Programming Large-scale Systems: State of the Art

Chapel at a Glance

Chapel: Base Language

Chapel: Data Parallelism

Chapel: Task Parallelism

Chapel: Locales

Chapel: Domain Maps
High-level Parallel Programming with Chapel

Programming Large-scale Systems: State of the Art

Chapel at a Glance

Chapel: Base Language

Chapel: Data Parallelism

Chapel: Task Parallelism

Chapel: Locales

Chapel: Domain Maps
High-level Parallel Programming with Chapel

Programming Large-scale Systems: State of the Art

Chapel at a Glance

Chapel: Base Language

Chapel: Data Parallelism

Chapel: Task Parallelism

Chapel: Locales

Chapel: Domain Maps
High-level Parallel Programming with Chapel

Programming Large-scale Systems: State of the Art

Chapel at a Glance

Chapel: Base Language

Chapel: Data Parallelism

Chapel: Task Parallelism

Chapel: Locales

Chapel: Domain Maps
Chapel: Background
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:**

\[
\begin{align*}
A & = \\
B & + \\
C & \cdot \\
\alpha &
\end{align*}
\]
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:
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;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    int rv, errCount;
    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;
    MPI_Comm comm = MPI_COMM_WORLD;

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

    int rv, errCount;
    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);

    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 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/UPC/CAF</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>OpenACC/CUDA/OpenCL</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…

MPI + OpenMP

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

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);
}

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

```c
#include <hpcc.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;
}
```

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

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;
```
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, ...
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

- **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”

\[
\begin{align*}
\text{Global-View} & \quad \text{Local-View} \\
( & \begin{array}{c}
\text{[Row of values]} \\
\end{array})/2 & \begin{array}{c}
\text{[Result]} \\
\end{array} \\
+ & \begin{array}{c}
\text{[Row of values]} \\
\end{array} & \begin{array}{c}
\text{[Result]} \\
\end{array} \\
= & \begin{array}{c}
\text{[Result]} \\
\end{array} & \begin{array}{c}
\text{[Result]} \\
\end{array}
\end{align*}
\]
2) Global-View Abstractions

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

\[ \text{Global-View} \]

\[
\left( \begin{array}{c}
\text{pink} \\
\text{yellow} \\
\text{pink}
\end{array} \right) + \left( \begin{array}{c}
\text{brown} \\
\text{yellow} \\
\text{brown}
\end{array} \right)/2 \\
= \left( \begin{array}{c}
\text{yellow} \\
\text{yellow} \\
\text{yellow}
\end{array} \right)
\]

\[ \text{Local-View} \]

\[
\left( \begin{array}{c}
\text{pink} \\
\text{yellow} \\
\text{pink}
\end{array} \right) + \left( \begin{array}{c}
\text{brown} \\
\text{yellow} \\
\text{brown}
\end{array} \right)/2 \\
= \left( \begin{array}{c}
\text{yellow} \\
\text{yellow} \\
\text{yellow}
\end{array} \right)
\]

\[
\left( \begin{array}{c}
\text{pink} \\
\text{yellow} \\
\text{pink}
\end{array} \right) + \left( \begin{array}{c}
\text{brown} \\
\text{yellow} \\
\text{brown}
\end{array} \right)/2 \\
= \left( \begin{array}{c}
\text{yellow} \\
\text{yellow} \\
\text{yellow}
\end{array} \right)
\]

\[
\left( \begin{array}{c}
\text{pink} \\
\text{yellow} \\
\text{pink}
\end{array} \right) + \left( \begin{array}{c}
\text{brown} \\
\text{yellow} \\
\text{brown}
\end{array} \right)/2 \\
= \left( \begin{array}{c}
\text{yellow} \\
\text{yellow} \\
\text{yellow}
\end{array} \right)
\]
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”

```
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(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?”
3) Multiresolution Design

**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
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
Chapel: Base Language
"Hello World" in Chapel: Two Versions

- Fast prototyping

```chapel
writeln("Hello, world!");
```

- "Production-grade"

```chapel
module Hello {
    proc main() {
        writeln("Hello, world!");
    }
}
```
C, Modula: basic syntax
ZPL, HPF: data parallelism, index sets, distributed arrays
CRA Y MTA C/Fortran: task parallelism, synchronization
CLU (see also Ruby, Python, C#): iterators
Scala (see also ML, Matlab, Perl, Python, C#): type inference
Java, C#: OOP, type safety
C++: generic programming/templates
   (but with a different syntax)
Outline

- Introductory Notes
- Elementary Concepts
  - Lexical structure
  - Types, variables, and constants
  - Operators and Assignments
  - Compound Statements
  - Input and output
- Data Types and Control Flow
- Program Structure
Lexical Structure

• Comments

```c
/* standard
   C style
   multi-line */
```

```cpp
// standard C++ style single-line
```

• Identifiers:
  • Composed of A-Z, a-z, _, $, 0-9
  • Cannot start with 0-9
  • Case-sensitive
### Primitive Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Default Value</th>
<th>Currently-Supported Bit Widths</th>
<th>Default Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>logical value</td>
<td>false</td>
<td>8, 16, 32, 64</td>
<td>impl. dep.</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
<td>0</td>
<td>8, 16, 32, 64</td>
<td>64</td>
</tr>
<tr>
<td>uint</td>
<td>unsigned integer</td>
<td>0</td>
<td>8, 16, 32, 64</td>
<td>64</td>
</tr>
<tr>
<td>real</td>
<td>real floating point</td>
<td>0.0</td>
<td>32, 64</td>
<td>64</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary floating point</td>
<td>0.0i</td>
<td>32, 64</td>
<td>64</td>
</tr>
<tr>
<td>complex</td>
<td>complex floating points</td>
<td>0.0 + 0.0i</td>
<td>64, 128</td>
<td>128</td>
</tr>
<tr>
<td>string</td>
<td>character string</td>
<td>&quot;&quot;</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Syntax

```
primitive-type: 
  type-name [( bit-width )]
```

#### Examples

```
int(16) // 16-bit int
real(32) // 32-bit real
uint // 64-bit uint
```
Implicit Type Conversions (Coercions)

- Notes:
  - reals do not implicitly convert to ints as in C
  - ints and uints don’t interconvert as handily as in C
Type Aliases and Casts

- **Basic Syntax**

  ```plaintext
type-alias-declaration:
    type identifier = type-expr;

cast-expr:
  expr : type-expr
```

- **Semantics**
  - type aliases are simply symbolic names for types
  - casts are supported between any primitive types

- **Examples**

  ```plaintext
type elementType = complex(64);

5:int(8) // store value as int(8) rather than int
"54":int // convert string to an int
249:elementType // convert int to complex(64)
```
**Variables, Constants, and Parameters**

- **Basic syntax**
  
  *declaration*:
  ```
  var identifier [: type] [= init-expr];
  const identifier [: type] [= init-expr];
  param identifier [: type] [= init-expr];
  ```

- **Semantics**
  - **var/const**: execution-time variable/constant
  - **param**: compile-time constant
  - No *init-expr* ⇒ initial value is the type’s default
  - No *type* ⇒ type is taken from *init-expr*

- **Examples**
  ```
  const pi: real = 3.14159;
  var count: int; // initialized to 0
  param debug = true; // inferred to be bool
  ```
Config Declarations

• Syntax

```
config-declaration:
    config type-alias-declaration
    config declaration
```

• Semantics

• Like normal, but supports command-line overrides
• Must be declared at module/file scope

• Examples

```
config param intSize = 32;
config type elementType = real(32);
config const epsilon = 0.01:elementType;
config var start = 1:int(intSize);
```

```
% chpl myProgram.chpl -sintSize=64 -selementType=real
% a.out --start=2 --epsilon=0.00001
```
## Basic Operators and Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
<th>Overloadable</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>cast</td>
<td>left</td>
<td>no</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>! ~</td>
<td>logical and bitwise negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>* / %</td>
<td>multiplication, division and modulus</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>unary + -</td>
<td>positive identity and negation</td>
<td>right</td>
<td>yes</td>
</tr>
<tr>
<td>+ -</td>
<td>addition and subtraction</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>shift left and shift right</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&lt;= &gt;= &lt; &gt;</td>
<td>ordered comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>== !=</td>
<td>equality comparison</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise/logical and</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>^</td>
<td>bitwise/logical xor</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>l</td>
<td>bitwise/logical or</td>
<td>left</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>short-circuiting logical and</td>
<td>left</td>
<td>via isTrue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>short-circuiting logical or</td>
</tr>
</tbody>
</table>
### Assignments

<table>
<thead>
<tr>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>simple assignment</td>
</tr>
<tr>
<td>+=  -=  *= /= %= **= &amp;=</td>
<td>= ^= &amp;&amp;=</td>
</tr>
<tr>
<td></td>
<td>(e.g., ( x += y ; ) is equivalent to ( x = x + y ; ))</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>swap assignment</td>
</tr>
</tbody>
</table>

- **Note**: assignments are only supported at the statement level
Compound Statements

• Syntax

\[
\text{compound-stmt:} \\
\{ \text{stmt-list} \}
\]

• Semantics
  • As in C, permits a series of statements to be used in place of a single statement

• Example

\[
\{ \\
\text{writeln(“Starting a compound statement”);} \\
x += 1; \\
\text{writeln(“Ending the compound statement”)}; \\
\}
\]
Output
- `write(expr-list)`: writes the argument expressions
- `writeln(...)`: writes a linefeed after the arguments

Input
- `read(expr-list)`: reads values into the argument expressions
- `read(type-list)`: reads values of given types, returns as tuple
- `readln(...)`: reads through next linefeed

Example:
```chapel
var first, last: string;
write("what is your name? ");
read(first);
last = read(string);
writeln("Hi ", first, " ", last);
```

I/O to files and strings also supported
Introductory Notes
Elementary Concepts
Data Types and Control Flow
  • Tuples
  • Ranges
  • Arrays
  • For loops
  • Other control flow
Program Structure
Tuples

- **Syntax**

  heterogeneous-tuple-type:
  
  \[( \text{type, type-list} )\]

  homogenous-tuple-type:
  
  \[\text{param-int-expr} \ast \text{type}\]

  tuple-expr:
  
  \[(\text{expr, expr-list})\]

- **Examples**

```
var coord: (int, int, int) = (1, 2, 3);
var coordCopy: 3*int = coord;
var (i1, i2, i3) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```

- **Purpose**

  - supports lightweight grouping of values
    (e.g., when passing or returning procedure arguments)
  - multidimensional arrays use tuple indices
Range Values

- **Syntax**
  
  `range-expr:
  [low] .. [high]`

- **Semantics**
  
  - Regular sequence of integers
    
    \( low \leq high: \) low, low+1, low+2, ..., high
    
    \( low > high: \) degenerate (an empty range)
    
    \( low \) or \( high \) unspecified: unbounded in that direction

- **Examples**
  
  1..6 // 1, 2, 3, 4, 5, 6
  6..1 // empty
  3.. // 3, 4, 5, 6, 7, ...
Range Operators

• Syntax

```
range-op-expr:
  range-expr by stride
  range-expr # count
  range-expr[range-expr]
```

• Semantics

- **by**: strides range; negative stride ⇒ start from high
- **#**: selects initial count elements of range
- **align**: specifies the alignment of a strided range
- [ ] or (): intersects the two ranges

• Examples

```
1..6 by 2   // 1, 3, 5
1..6 by -1  // 6, 5, 4, ..., 1
1..6 # 4    // 1, 2, 3, 4
1..6[3..]   // 3, 4, 5, 6
```

```
1.. by 2     // 1, 3, 5, ...
1.. by 2 #3   // 1, 3, 5
1.. by 2 align 2 // 2, 4, ...
1.. #3 by 2   // 1, 3
0..#n         // 0, ..., n-1
```
Array Types

• Syntax

array-type:
[ index-set-expr ] elt-type

• Semantics

• Stores an element of elt-type for each index
• Array values expressed using square brackets

• Examples

var A: [1..3] int = [5, 3, 9], // 3-element array of ints
B: [1..3, 1..5] real, // 2D array of reals
C: [1..3][1..5] real; // array of arrays of reals

Much more on arrays in data parallelism section later...
For Loops

- **Syntax**
  
  ```
  for-loop:
  for index-exp in iterable-exp { stmt-list }
  ```

- **Semantics**
  
  - Executes loop body serially, once per loop iteration
  - Declares new variables for identifiers in `index-exp`
    - type and const-ness determined by `iteratable-exp`
    - `iteratable-exp` could be a range, array, or iterator
  
- **Examples**
  
  ```
  var A: [1..3] string = [" DO", " RE", " MI"]; 
  
  for i in 1..3 { write(A(i)); } // DO RE MI 
  for a in A { a += "LA"; } write(A); // DOLA RELA MILA
  ```
Zipper Iteration

• Syntax

    zipper-for-loop:
    for indexExpr in zip( iterableExprs ) { stmt-list }

• Semantics

  • Zipper iteration is over all yielded indices pair-wise

• Example

    var A: [0..9] real;

    for (a,i,j) in zip(A, 1..10, 2..20 by 2) do
      a = j + i/10.0;

    writeln(A);

    2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
Other Control Flow Statements

- Conditional statements
  ```
  if cond { computeA(); } else { computeB(); }
  ```

- While loops
  ```
  while cond {
    compute();
  }
  ```

- Select statements
  ```
  select key {
    when value1 { compute1(); }
    when value2 { compute2(); }
    otherwise { compute3(); }
  }
  ```

Note: Chapel also has expression-level conditionals and for loops
Most control flow supports keyword-based forms for single-statement versions

- **Conditional statements**
  ```
  if cond then computeA(); else computeB();
  ```

- **While loops**
  ```
  while cond do compute();
  ```

- **For loops**
  ```
  for indices in iterable-expr do compute();
  ```

- **Select statements**
  ```
  select key {
  when value1 do compute1();
  when value2 do compute2();
  otherwise do compute3();
  }
  ```
Outline

• Introductory Notes
• Elementary Concepts
• Data Types and Control Flow
• Program Structure
  • Procedures and iterators
  • Modules and main()
  • Records and classes
  • Generics
  • Other basic language features
Procedures, by example

- Example to compute the area of a circle

```
proc area(radius: real): real {
    return 3.14 * radius**2;
}

writeln(area(2.0));  // 12.56
```

- Example of argument default values, naming

```
proc writeCoord(x: real = 0.0, y: real = 0.0) {
    writeln((x,y));
}

writeCoord(2.0);      // (2.0, 0.0)
writeCoord(y=2.0);    // (0.0, 2.0)
writeCoord(y=2.0, 3.0);  // (3.0, 2.0)
```
Iterators

**Iterator:** a procedure that generates values/variables
- Used to drive loops or populate data structures
- Like a procedure, but yields values back to invocation site
- Control flow logically continues from that point

**Example**

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

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

0
1
1
2
3
5
8
Argument and Return Intents

• Arguments can optionally be given intents
  • (blank): varies with type; follows principle of least surprise
    • most types: const
    • arrays, domains, sync vars: passed by reference
  • const: disallows modification of the formal
  • in: copies actual into formal at start; permits modifications
  • out: copies formal into actual at procedure return
  • inout: does both of the above
  • ref: pass by reference
  • param/type: formal must be a param/type (evaluated at compile-time)

• Return types can also have intents
  • (blank)/const: cannot be modified (without assigning to a variable)
  • var: permits modification back at the callsite
  • type: returns a type (evaluted at compile-time)
  • param: returns a param value (evaluated at compile-time)
Modules

**Syntax**

```plaintext
module-def:
    module identifier { code }

module-use:
    use module-identifier;
```

**Semantics**

- all Chapel code is stored in modules
- `use`-ing a module makes its symbols visible in that scope
- module-level statements are executed at program startup
  - typically used to initialize the module
- for convenience, a file containing code outside of a module declaration creates a module with the file’s name
• Semantics
  • Chapel programs start by:
    • initializing all modules
    • executing main(), if it exists

M1.chpl:
use M2;
writeln("Initializing M1");
proc main() { writeln("Running M1"); }

M2.chpl:
module M2 {
  writeln("Initializing M2");
}

% chpl M1.chpl M2.chpl
% ./a.out
Initializing M2
Initializing M1
Running M1
Revisiting "Hello World"

- Fast prototyping

```plaintext
hello.chpl

writeln("Hello, world!");
```

- "Production-grade"

```plaintext
module HelloWorld {
  proc main() {
    writeln("Hello, world!");
  }
}
```

- Module-level code is executed during module initialization
- `main()` executed when program begins running
Records and Classes

• Chapel’s struct/object types
  • Contain variable definitions (fields)
  • Contain procedure & iterator definitions (methods)
  • Records: value-based (e.g., assignment copies fields)
  • Classes: reference-based (e.g., assignment aliases object)
  • Record : Class :: C++ struct : Java class

• Example

```plaintext
record circle {
  var radius: real;
  proc area() {
    return pi*radius**2;
  }
}

var c1, c2: circle;
c1 = new c1(radius=1.0);
c2 = c1; // copies c1
c1.radius = 5.0;
writeln(c2.radius); // 1.0
// records deleted by compiler
```
Chapel’s struct/object types
- Contain variable definitions (fields)
- Contain procedure & iterator definitions (methods)
- Records: value-based (e.g., assignment copies fields)
- Classes: reference-based (e.g., assignment aliases object)

Record : Class :: C++ struct : Java class

Example

```plaintext
class circle {
    var radius: real;
    proc area() {
        return pi*radius**2;
    }
}
```

```plaintext
var c1, c2: circle;
c1 = new c1(radius=1.0);
c2 = c1; // aliases c1’s circle
c1.radius = 5.0;
writeln(c2.radius); // 5.0
delete c1; // users delete classes
```
Methods without arguments need not use parenthesis

```plaintext
proc circle.circumference {
    return 2* pi * radius;
}
writeln(c1.area(), " ", c1.circumference);
```

Methods can be defined for any type

```plaintext
proc int.square() {
    return this**2;
}
writeln(5.square());
```
Chapel: Data Parallelism
Data Parallelism:

- parallelism is driven by collections of data
  - data aggregates (arrays)
  - sets of indices (ranges, domains)
  - other user-defined collections
- e.g., “for all elements in array A …”

Task Parallelism:

- parallelism is expressed in terms of distinct computations
- e.g., “create a task to do foo() while another does bar()”

(Of course, data parallelism is executed using tasks and task parallelism typically operates on data, so the line can get fuzzy at times…)

"Hello World" in Chapel: a Data Parallel Version

• Data Parallel Hello World

```chapel
config const numIters = 100000;

forall i in 1..numIters do
    writeln("Hello, world! ",
            "from iteration ", i,
            " of ", numIters);
```
Outline

- Domains and Arrays
  - Rectangular Domains and Arrays
  - Iterations and Operations
- Other Domain Types
- Reductions and Scans
- Jacobi Iteration Example
Domain: A first-class index set

- A fundamental Chapel concept for data parallelism
- Domains may optionally be distributed
config const \( m = 4, \ n = 8 \);

\textbf{var} \( D: \text{domain}(2) = \{1..m, 1..n\} \);
config const m = 4, n = 8;

var D: domain(2) = {1..m, 1..n};

var Inner: subdomain(D) = {2..m-1, 2..n-1};
Domains Define Arrays

- **Syntax**

  \[
  \text{array-type:} \\
  [ \text{domain-expr} ] \text{elt-type}
  \]

- **Semantics**
  - Stores an \textit{elt-type} for each index in \textit{domain-expr}

- **Example**

  \[
  \text{var } A, B: [D] \text{ real;}
  \]

- **Earlier example, revisited**

  \[
  \text{var } A: [1..3, 1..5] \text{ real;} // [1..3, 1..5] \text{ creates an anonymous domain}
  \]
Domain Iteration

- **For loops (discussed already)**
  - Execute loop body once per domain index, serially

```plaintext
for i in Inner do ...
```

- **Forall loops**
  - Executes loop body once per domain index, in parallel
  - Loop must be *serializable* (executable by one task)

```plaintext
forall i in Inner do ...
```

- Loop variables take on `const` domain index values

![Diagram of domain iteration](image)
Other Forall Loops

Forall loops also support...

- A shorthand notation:

\[
\begin{align*}
[(i,j) \text{ in } D] \quad & A[i,j] = i + j/10.0; \\
A &= \forall (i,j) \text{ in } D \quad \text{do} \quad i + j/10.0; \\
A &= [(i,j) \text{ in } D] \quad i + j/10.0;
\end{align*}
\]

- Expression-based forms:
Domain values support...

- **Methods for creating new domains**
  
  ```
  var D2 = Inner.expand(1, 0);
  ```

  ```
  var D3 = Inner.translate(0, 1);
  ```

- **Intersection via Slicing**
  
  ```
  var D4 = D2[D3];
  ```

- **Range operators (e.g., #, by, align)**
Indexing into arrays with domain values results in a sub-array expression (an “array slice”)

```c
A[Inner] = B[Inner.translate(0,1)];
```

![Diagram](image)
Array Reallocation

Reassigning a domain logically reallocates its arrays
• array values are preserved for common indices

\[ D = \{1..2m, 1..2n\}; \]
Array Iteration

- Array expressions also support for and forall loops

```plaintext
for a in A[Inner] do ...
```

```
forall a in A[Inner] do ...
```

- Array loop indices refer to array elements (can be modified)

```plaintext
forall (a, (i,j)) in zip(A, D) do a = i + j/10.0;
```

Note that forall loops support zippered iteration, like for-loops
Array Indexing

• Arrays can be indexed using variables of their domain’s index type (tuples) or lists of integers

```plaintext
var i = 1, j = 2;
var ij = (i, j);
A[ij] = 1.0;
A[i, j] = 2.0;
```

• Array indexing can use either parentheses or brackets

```plaintext
A(ij) = 3.0;
A(i, j) = 4.0;
```
Array Arguments and Aliases

- Arrays are passed by reference by default

```plaintext
class proc zero(X: []) { X = 0; }
zero(A[Inner]); // zeroes the inner values of A
```

- Formal array arguments can reindex actuals

```plaintext
class proc f(X: [1..b,1..b]) { ... } // X uses 1-based indices
f(A[lo..#b, lo..#b]);
```

- Array alias declarations provide similar functionality

```plaintext
class var InnerA => A[Inner];
class var InnerA1: [1..n-2,1..m-2] => A[2..n-1,2..m-1];
```
Promoting Functions and Operators

Functions/operators expecting scalars can also take...
...arrays, causing each element to be passed in

\[
\begin{align*}
\sin(A) & \approx \text{forall } a \in A \text{ do } \sin(a) \\
2*A & \approx \text{forall } a \in A \text{ do } 2*a
\end{align*}
\]

...domains, causing each index to be passed in

\[
\begin{align*}
\text{foo}(\text{Inner}) & \approx \text{forall } i \in \text{Inner} \text{ do } \text{foo}(i)
\end{align*}
\]

Multiple arguments promote using zippered iteration

\[
\begin{align*}
\text{pow}(A, B) & \approx \text{forall } (a,b) \in \text{zip}(A,B) \text{ do } \text{pow}(a,b)
\end{align*}
\]
Data Parallelism is Implicit

- forall loops are implemented using multiple tasks
  - ditto for operations that are equivalent to foralls
  - details depend on what is being iterated over

- many times, this parallelism can seem invisible
  - for this reason, Chapel’s data parallelism can be considered *implicitly parallel*
  - it also tends to make the data parallel features easier to use and less likely to result in bugs as compared to explicit tasks
By default*, controlled by three config variables:

--dataParTasksPerLocale=#
  • Specify # of tasks to execute forall loops
  • Current Default: number of processor cores

--dataParIgnoreRunningTasks=[true | false]
  • If false, reduce # of forall tasks by # of running tasks
  • Current Default: true

--dataParMinGranularity=#
  • If > 0, reduce # of forall tasks if any task has fewer iterations
  • Current Default: 1

*Default values can be overridden for specific domains/arrays
Outline

- Domains and Arrays
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Reductions and Scans
- Jacobi Iteration Example
Chapel Domain Types

Chapel supports several domain types...

```chapel
var OceanSpace = {0..#lat, 0..#long},
AirSpace = OceanSpace by (2,4),
IceSpace: sparse subdomain(OceanSpace) = genCaps();
```

![Domain Types Diagram]

- **dense**
- **strided**
- **sparse**
- **unstructured**
- **associative**

```chapel
var Vertices: domain(opaque) = ...,
People: domain(string) = ...;
```
All domain types can be used to declare arrays...  

```chapel
var Ocean: [OceanSpace] real,
   Air: [AirSpace] real,
   IceCaps[IceSpace] real;
```

```chapel
var Weight: [Vertices] real,
   Age: [People] int;
```
...to iterate over index sets...

```latex
forall \ ij \ in \ AirSpace \ do
\hspace{1cm} \text{Ocean}[ij] += \text{IceCaps}[ij];
```

```latex
forall \ v \ in \ Vertices \ do
\hspace{1cm} \text{Weight}[v] = \text{numEdges}[v];
```

```latex
forall \ p \ in \ People \ do
\hspace{1cm} \text{Age}[p] += 1;
```
Slicing

...to slice arrays...

Ocean[AirSpace] += IceCaps[AirSpace];

...Vertices[Interior]...

...People[Interns]...
Reallocation

...and to reallocate arrays

AirSpace = OceanSpace by (2,2);
IceSpace += genEquator();

newnode = Vertices.create();
People += "vass";
var Presidents: domain(string) =
    {“George”, “John”, “Thomas”,
     “James”, “Andrew”, “Martin”};

Presidents += “William”;

var Age: [Presidents] int,
    Birthday: [Presidents] string;

Birthday[“George”] = “Feb 22”;

forall president in President do
    if Birthday[president] == today then
        Age[president] += 1;
• Domains and Arrays
• Other Domain Types
• Reductions and Scans
• Jacobi Iteration Example
Reductions

• Syntax

\[
\text{reduce-expr:} \\
\text{reduce-op reduce iterator-expr}
\]

• Semantics

• Combines argument values using \textit{reduce-op}

• \textit{Reduce-op} may be built-in or user-defined

• Examples

\[
\begin{align*}
\text{total} &= + \ \text{reduce} \ A; \\
\text{bigDiff} &= \ \text{max reduce} \ [i \ \text{in} \ \text{Inner}] \ \text{abs}(A[i]-B[i]); \\
\text{(minVal, minLoc)} &= \ \text{minloc reduce zip}(A, \ D);
\end{align*}
\]
### Syntax

\[
\text{scan-expr:} \\
\text{scan-op scan iterator-expr}
\]

### Semantics

- Computes parallel prefix over values using \textit{scan-op}
- \textit{Scan-op} may be any \textit{reduce-op}

### Examples

```chapel
var A, B, C: [1..5] int;
A = 1;       // A: 1 1 1 1 1
B = + scan A; // B: 1 2 3 4 5
C = min scan B; // C: 1 1 -3 -3 -3
```
Reduction and Scan Operators

- **Built-in**
  - +, *, &&, | |, &, |, ^, min, max
  - minloc, maxloc
    - Takes a zipped pair of values and indices
    - Generates a tuple of the min/max value and its index

- **User-defined**
  - Defined via a class that implements a standard interface
  - Compiler generates code that calls these methods
Outline

- Domains and Arrays
- Other Domain Types
- Reductions and Scans
- Jacobi Iteration Example
Jacobi Iteration in Pictures

\[ A: \sum \frac{1}{4} \]  

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 {
                           + 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` cannot change values after initialization

- `n` is a default integer (32 bits)
- `epsilon` is a 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;
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;
    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
} while (delta > epsilon);
writeln(A);
```

Declare domains (first class index sets)

- `domain(2)` ⇒ 2D arithmetic domain, indices are integer 2-tuples
- `subdomain(P)` ⇒ a domain of the same type as `P` whose indices are guaranteed to be a subset of `P`'s
- `exterior` ⇒ one of several built-in domain generators
Jacobi Iteration in 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** \(\Rightarrow\) can be modified throughout its lifetime

**: [BigD] T** \(\Rightarrow\) array of size BigD with elements of type T

**(no initializer)** \(\Rightarrow\) 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

```
+---+---+---+---+---+---+
|   |   |   |   |   |   |
+---+---+---+---+---+---+
|   |   |   |   |   |   |
+---+---+---+---+---+---+
|   |   |   |   |   |   |
+---+---+---+---+---+---+
|   |   |   |   |   |   |
+---+---+---+---+---+---+
```

```
  1  1  1  1  1  1
  1  1  1  1  1  1
  1  1  1  1  1  1
  1  1  1  1  1  1
  1  1  1  1  1  1
  1  1  1  1  1  1
```
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 {
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
```

Compute 5-point stencil

$\sum \left( \begin{array}{c}
    \text{orange} \\
    \text{yellow} \\
    \text{orange}
\end{array} \right) / 4$


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

writeln($A$);
Jacobi Iteration in Chapel

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

cfg 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

```chapel
do {

    cfg 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 {
     + 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 {
    + 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
Chapel: Task Parallelism
Task Creation: Begin

- **Syntax**
  
  \[
  \text{begin-stmt:}
  \begin{align*}
  \text{begin} & \quad \text{stmt}
  \end{align*}
  \]

- **Semantics**
  
  - Creates a task to execute \textit{stmt}
  
  - Original (“parent”) task continues without waiting

- **Example**

  \[
  \text{begin writeln("hello world");}
  \text{writeln("good bye");}
  \]

- **Possible output**

  hello world  
good bye  
good bye  
hello world
Block-Structured Task Creation: Cobegin

- **Syntax**
  
  ```cobegin-stmt:
  cobegin { stmt-list }
  ```

- **Semantics**
  - Creates a task for each statement in `stmt-list`
  - Parent task waits for `stmt-list` tasks to complete

- **Example**
  
  ```
  cobegin {
    foo(1);
    foo(2);
    bar();
  }  // wait here for both foo()s and bar() to return
  ```
Loop-Structured Task Invocation: Coforall

- **Syntax**

  ```
  coforall-loop:
  coforall index-expr in iterable-expr { stmt-list }
  ```

- **Semantics**

  - Create a task for each iteration in `iteratable-expr`
  - Parent task waits for all iteration tasks to complete

- **Example**

  ```
  config const numTasks = here.numCores;
  
  coforall tid in 0..#numTasks do
  writeln("Hello, world!",
          "from task ", tid, " of ", numTasks);
  ```
begin:
- Use to create a dynamic task with an unstructured lifetime
- “fire and forget”

cobegin:
- Use to create a related set of heterogeneous tasks
- ...or a small, finite set of homogenous tasks
- The parent task depends on the completion of the tasks

coforall:
- Use to create a fixed or dynamic # of homogenous tasks
- The parent task depends on the completion of the tasks

Note: All these concepts can be composed arbitrarily
Joining Sub-Tasks: Sync-Statements

• Syntax

```plaintext
sync-statement:
  sync stmt
```

• Semantics

• Executes `stmt`
• Waits for all `dynamically-scoped` begins to complete

• Example

```plaintext
sync 
  { 
    for i in 1..numFoos 
      { 
        begin foo(i);
      } 
    bar();
  }

proc search(N: TreeNode) 
  { 
    if (N != nil) 
      { 
        begin search(N.left);
        begin search(N.right);
      } 
    }

sync { search(root); }
```
Sync-Statements and Dynamic Scoping

Where the cobegin statement is static...

```cobegin
{ 
    functionWithBegin();
    functionWithoutBegin();
} // waits on these two tasks, but not any others
```

...the sync statement is dynamic.

```sync
{ 
    begin functionWithBegin();
    begin functionWithoutBegin();
} // waits on these tasks and any other descendents
```
Sync-Statements and Program Termination

Program termination is defined by an implicit sync on the main() procedure:

```c
sync main();
```
Chapel: Locales

(Controlling Locality and Affinity)
The Locale Type

Definition:
- Abstract unit of target architecture
- Supports reasoning about locality
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Properties:
- a locale’s tasks have ~uniform access to local vars
- Other locale’s vars are accessible, but at a price

In practice:
- Typically a compute node (multicore processor or SMP)
"Hello World" in Chapel: a Multi-Locale Version

- Multi-locale Hello World

```chapel
coforall loc in Locales do
  on loc do
    writeln("Hello, world! ",
            "from node ", loc.id,
            " of ", numLocales);
```
Locales and Program Startup

- Specify # of locales when running Chapel programs

\[
\% \text{a.out --numLocales=8} \quad \% \text{a.out -nl 8}
\]

- Chapel provides built-in locale variables

```chapel
config const numLocales: int = ...;
const LocaleSpace = {0..numLocales-1};
const Locales: [LocaleSpace] locale = ...;
```

- main() begins as a single task on locale #0 (Locales[0])
Rearranging Locales

Create locale views with standard array operations:

```plaintext
var TaskALocs = Locales[0..1];
var TaskBLocs = Locales[2..];
var Grid2D = reshape(Locales, {1..2, 1..4});
```

<table>
<thead>
<tr>
<th>Locales:</th>
<th>L0</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
<th>L6</th>
<th>L7</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskALocs:</td>
<td>L0</td>
<td>L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskBLocs:</td>
<td>L2</td>
<td>L3</td>
<td>L4</td>
<td>L5</td>
<td>L6</td>
<td>L7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grid2D:</td>
<td>L0</td>
<td>L1</td>
<td>L2</td>
<td>L3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L4</td>
<td>L5</td>
<td>L6</td>
<td>L7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Locale Methods

- **proc locale.id: int { ... }**
  Returns locale’s index in LocaleSpace

- **proc locale.name: string { ... }**
  Returns name of locale, if available (like uname -a)

- **proc locale.numCores: int { ... }**
  Returns number of processor cores available to locale

- **proc locale.physicalMemory(...) { ... }**
  Returns physical memory available to user programs on locale

**Example**

```plaintext
const totalPhysicalMemory =
+ reduce Locales.physicalMemory();
```
The On Statement

- **Syntax**
  
  \[
  \text{on-stmt:}
  \]
  \[
  \quad \text{on expr do stmt}
  \]
  \[
  \quad \text{on expr \{ stmts \}}
  \]

- **Semantics**
  - Executes \textit{stmt(s)} on the locale that stores \textit{expr}

- **Example**
  
  writeln(“start executing on locale 0”); 
  on Locales[1] do 
  \[
  \quad \text{writeln(“now we’re on locale 1”);}
  \]
  writeln(“back on locale 0 again”);
Locality and Parallelism are Orthogonal

- On-clauses do not introduce any parallelism

```plaintext
writeln(“start executing on locale 0”);
on Locales[1] do
    writeln(“now we’re on locale 1”);
writeln(“back on locale 0 again”);
```

- But can be combined with constructs that do:

```plaintext
writeln(“start executing on locale 0”);
cobegin {
    on Locales[1] do
        writeln(“this task runs on locale 1”);
    on Locales[2] do
        writeln(“while this one runs on locale 2”);
}
writeln(“back on locale 0 again”);
```

- Orthogonal support for parallelism and locality is key
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) {
    ...
}
```
Data-driven on-clauses

- On-clauses can also use a data-driven form...

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

...supporting affinity between tasks and their data

*(Note that even the ‘on Locales[3]’ form can be considered data-driven, since each locale stores its respective locale value)*
Q: How does data get onto other locales to begin with?

A1: Lexical scoping

```plaintext
var x: int;       // x is stored on locale 0
on Locales[1] {
    var y: int;   // y is stored on locale 1
    on Locales[2] {
        var z: int;   // z is stored on locale 2

        on y { y -= 1; }  // executes on locale 1
    }
}
```

<table>
<thead>
<tr>
<th>Loc 0</th>
<th>Loc 1</th>
<th>Loc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>
Q: How does data get onto other locales to begin with?

A2: Class instances

class C { var x, y, z: real; var next: C; }

var myC: C; // myC is stored on locale 0

on Locales[1] {
    myC = new C(...); // myC’s object lives on locale 1...
    on Locales[2] do
        myC.next = new C(...); // and its next is on locale 2
    }

on myC do ... // executes on locale 1
on myC.next do ... // executes on locale 2
Q: How does data get onto other locales to begin with?

A3: On-declarations (not yet implemented)

```plaintext
on Locales[1] var x: real;  // x is stored on locale 1
on Locales[2] var y: real;  // y is stored on locale 2

on x do ...  // executes on locale 1
on y do ...  // executes on locale 2
```
Q: How does data get onto other locales to begin with?

A4: Distributed domains and arrays (next slide deck)
Querying a Variable's Locale

- **Syntax**
  ```plaintext
  locale-query-expr: 
  expr . locale
  ```

- **Semantics**
  - Returns the locale on which `expr` is stored

- **Example**
  ```plaintext
  var i: int;
  on Locales[1] { 
    var j: int;
    writeln((i.locale.id, j.locale.id));  // outputs (0,1)
  }
  ```
• **Built-in locale variable**

```
const here: locale;
```

• **Semantics**

• Refers to the locale on which the task is executing

• **Example**

```
writeln(here.id);     // outputs 0
on Locales[1] do
  writeln(here.id);  // outputs 1

on myC do
  if (here == Locales[0]) then ...
```
• Without optimizations, Chapel’s global address space implies implicit communication

```chapel
var x: int;

on Locales[1] { // on-clause implies an active message
    var y: int;
    y = x; // implies a remote get of x
    on x do
        y = x; // implies a remote put to y
    }
```
The compiler can optimize communication subject to Chapel’s memory consistency model

```plaintext
var x: int;

on Locales[1] {  // on-clause implies an active message
    var y: int;
    y = x;  // in practice, read-only values like x
    // are bundled with the active message
}
```
Local statement

• Syntax

local-stmt:
  local { stmt };

• Semantics

  • Asserts to the compiler that all operations are local

• Example

on Locales[1] {  
  var myC: C = ...;
  ...
  myC.x += 1; // is myC.x local?
}

on Locales[1] {  
  var myC: C = ...;
  ...
  local { // assert it is
    myC.x += 1;
  }
}

• Note: Our current hope is to deprecate this feature, replacing it with data-centric concepts
• Most everything works correctly
  • exception: the on-declaration syntactic form
• The compiler is currently conservative about assuming variables may be remote
  • Impact: scalar performance overhead
• The compiler is currently lacking several important communication optimizations
  • Impact: scalability tends to be limited for programs with structured communication
Chapel: Domain Maps

(Layouts and Distributions)
Multi-locale Data Parallel Hello World

```chapel
class config const numIters = 100000;
const WorkSpace = {1..numIters} dmapped Block(...);

forall i in WorkSpace do
    writeln("Hello, world! ",
              "from iteration ", i,
              " of ", numIters,
              " on locale ", here.id,
              " of ", numLocales);
```

"Hello World" in Chapel: a Domain-Map Version
Domains are first-class index sets
- Specify the size and shape of arrays
- Support iteration, array operations, etc.

\[ \text{InnerD} \]

\[
\begin{array}{c}
D \\
A \\
B
\end{array}
\]
Data Parallelism: Implementation Qs

Q1: How are arrays laid out in memory?
   - Are regular arrays laid out in row- or column-major order? Or...?
   - What data structure is used to store sparse arrays? (COO, CSR, ...?)

Q2: How are data parallel operators implemented?
   - How many tasks?
   - How is the iteration space divided between the tasks?
Q3: How are arrays distributed between locales?
  - Completely local to one locale? Or distributed?
  - If distributed... In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? ...?

Q4: What architectural features will be used?
  - Can/Will the computation be executed using CPUs? GPUs? both?
  - What memory type(s) is the array stored in? CPU? GPU? texture? ...?

A1: In Chapel, any of these could be the correct answer

A2: Chapel’s domain maps are designed to give the user full control over such decisions
Domain maps are “recipes” (written in Chapel) that instruct the compiler how to map the global view of a computation...

\[ A = B + \alpha \cdot C; \]

...to the target locales’ memory and processors:
**Domain Maps:** “recipes for implementing parallel/distributed arrays and domains”

They define data storage:
- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in each locale’s memory

...as well as operations:
- random access, iteration, slicing, reindexing, rank change, ...
- the Chapel compiler generates calls to these methods to implement the user’s array operations
\texttt{const ProblemSpace} = \{1..m\};

\texttt{var A, B, C: [ProblemSpace] real;}

\texttt{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}

\[ \text{dmapped} \text{ } \text{Block} \text{ (boundingBox} = \{1..m\}) ; \]

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

\[ A = B + \alpha \cdot C ; \]
```
const ProblemSpace = {1..m}!
dmapped Cyclic(startIdx=1);

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

A = B + alpha * C;
```
Domain Maps fall into two major categories:

**layouts:** target a single locale
- (that is, a desktop machine or multicore node)
- **examples:** row- and column-major order, tilings, compressed sparse row

**distributions:** target multiple locales
- (that is, a distributed memory cluster or supercomputer)
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, ...
Declaring a Distributed Domain

• Domain types and literals may be domain mapped
  • In practice, this tends to be a great place to rely on type inference to avoid repetition:

    ```
    const Dom = {1..m, 1..n} dmapped myDMap(...);
    ```

• Domain maps can also be declared independently of a domain value (not covered here)
  • Useful for declaring several domains using the same map
var Dom = {1..4, 1..8} dmapped Block(boundingBox={1..4, 1..8});

distributed to

var Dom = {1..4, 1..8} dmapped Cyclic(startIdx=(1,1));

distributed to
The Block class constructor

```
proc Block(boundingBox: domain,
            targetLocales: [] locale = Locales,
            dataParTasksPerLocale = ...,
            dataParIgnoreRunningTasks = ...,
            dataParMinGranularity = ...)
```
The Cyclic class constructor

```plaintext
proc Cyclic(startIdx,
    targetLocales: [] locale = Locales,
    dataParTasksPerLocale = ...,
    dataParIgnoreRunningTasks = ...,
    dataParMinGranularity = ...) distributed to L0 L1 L2 L3 L4 L5 L6 L7
```
Having applied a domain map to a domain/array...

```plaintext
const Dom = {1..m, 1..n} dmapped Block(...);
var A, B: [Dom] real;
```

forall loops over that domain and array will be distributed according to the domain map

- i.e., data parallel expressions like:

```plaintext
forall ij in Dom do ...;
forall a in A do ...;
B = sin(A);
```

- result in code like:

```plaintext
coforall loc in <Dom's domainMap>.targetLocales do
  on loc do
    forall ij in <local portion of Dom, A, B>, ... do ...
```
All Domain Types Support Domain Maps

- **dense**
- **strided**
- **sparse**
- **unstructured**
- **associative**
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 the 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
Jacobi Iteration in Chapel

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

cost 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;
```

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 {
        + 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);
```