

Chapel Comes of Age: A Language for Productivity, Parallelism, and Performance

Brad Chamberlain

HPC Knowledge Meeting '19

June 14, 2019

✉ bradc@cray.com
🌐 chapel-lang.org
🐦 @ChapelLanguage



CRAY®



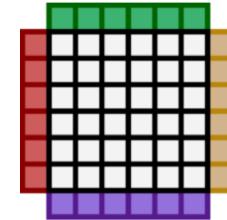
Who am I?

CRAY

Education:



- Earned Ph.D. from University of Washington CSE in 2001
 - focused on the ZPL data-parallel array language
- Remain associated with UW CSE as an Affiliate Professor



Industry:

CRAY

- A Principal Engineer at Cray Inc.
- The technical lead / a founding member of the Chapel project



What is Chapel?

CRAY

Chapel: A modern parallel programming language

- portable & scalable
- open-source & collaborative

Goals:

- Support general parallel programming
 - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive



What does “Productivity” mean to you?

CRAY

Recent Graduates:

“something similar to what I used in school: Python, Matlab, Java, ...”

Seasoned HPC Programmers:

“that sugary stuff that I don’t need because I ~~was born to suffer~~”

want full control to ensure performance”

Computational Scientists:

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

Chapel Team:

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

Why Consider New Languages at all?

CRAY

Syntax

- High level, elegant syntax
- Improve programmer productivity

Semantics

- Static analysis can help with correctness
- We need a compiler (front-end)

Performance

- If optimizations are needed to get performance
- We need a compiler (back-end)

Algorithms

- Language defines what is easy and hard
- Influences algorithmic thinking

[Source: Kathy Yelick,
CHI UW 2018 keynote:
*Why Languages Matter
More Than Ever*]

Outline

- ✓ Context and Motivation
- Chapel and Productivity
 - A Brief Tour of Chapel Features
 - Arkouda: NumPy over Chapel
 - Summary and Resources



Comparing Chapel to Other Languages

CRAY

Chapel aims to be as...

...**programmable** as Python

...**fast** as Fortran

...**scalable** as MPI, SHMEM, or UPC

...**portable** as C

...**flexible** as C++

...**fun** as [your favorite programming language]

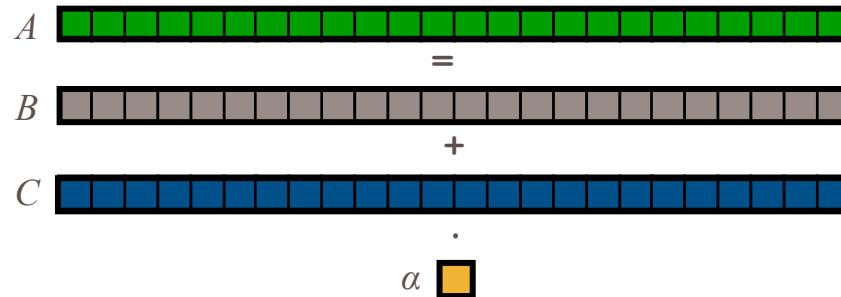
STREAM Triad: a trivial parallel computation

CRAY

Given: m -element vectors A, B, C

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

In pictures:



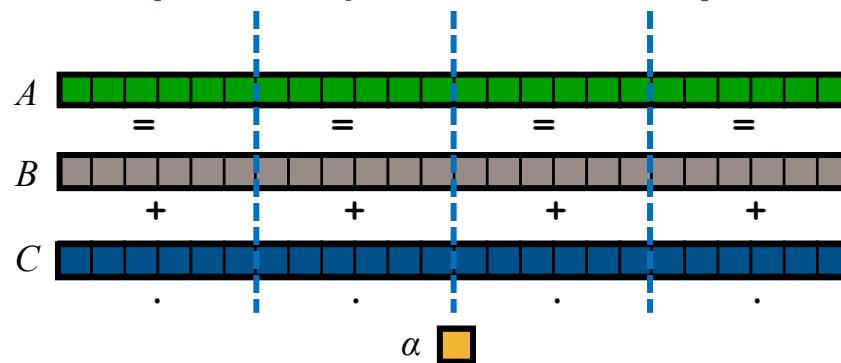
STREAM Triad: a trivial parallel computation

CRAY

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 (shared memory / multicore):



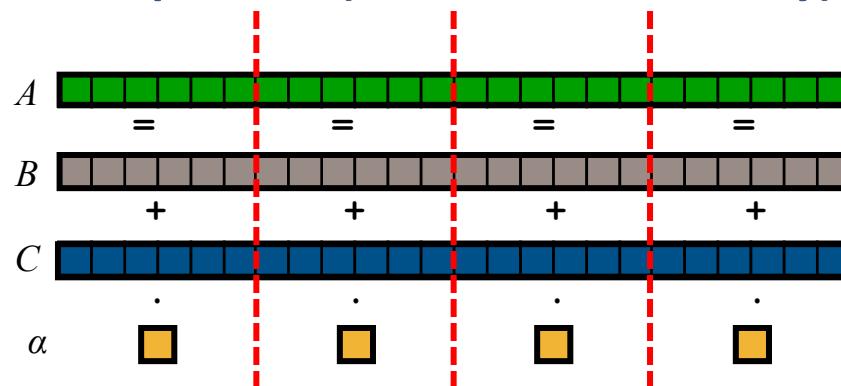
STREAM Triad: a trivial parallel computation

CRAY

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):



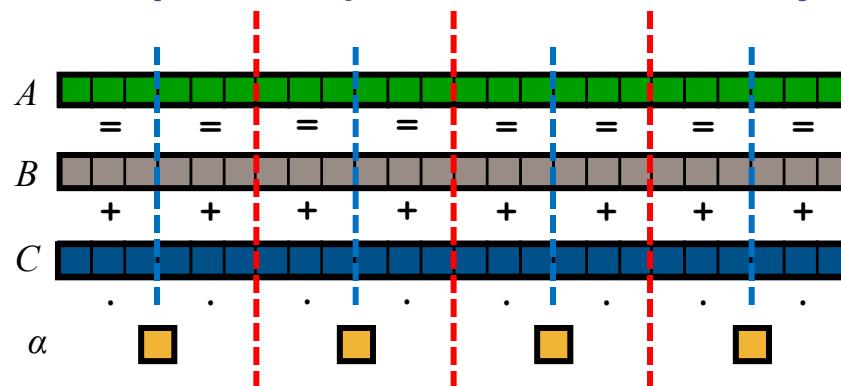
STREAM Triad: a trivial parallel computation

CRAY

Given: m -element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore):



STREAM Triad: C + MPI

CRAY

```
#include <hpcc.h>

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

int HPCC_StarStream(HPCC_Parms *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_Parms *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] = 1.0;
    }
    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;
}
```

STREAM Triad: C + MPI + OpenMP



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

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

int HPCC_StarStream(HPCC_Parms *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_Parms *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;
    }

#ifndef _OPENMP
#pragma omp parallel for
#endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }
    scalar = 3.0;

#ifndef _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;
}
```

STREAM Triad: Chapel

CRAY

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

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

int HPCC_StarStream(HPCC_Parms *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_IN
```

```
use ...;

config const m = 1000,
        alpha = 3.0;

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

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

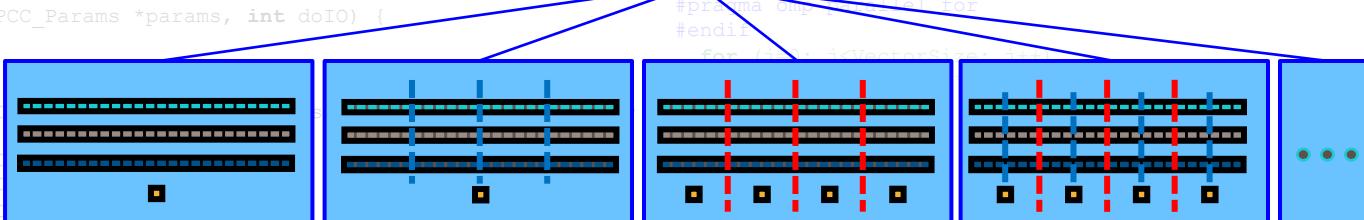
B = 2.0;
C = 1.0;

A = B + alpha * C;
```

The special sauce:
How should this index set—and the arrays and computations over it—be mapped to the system?

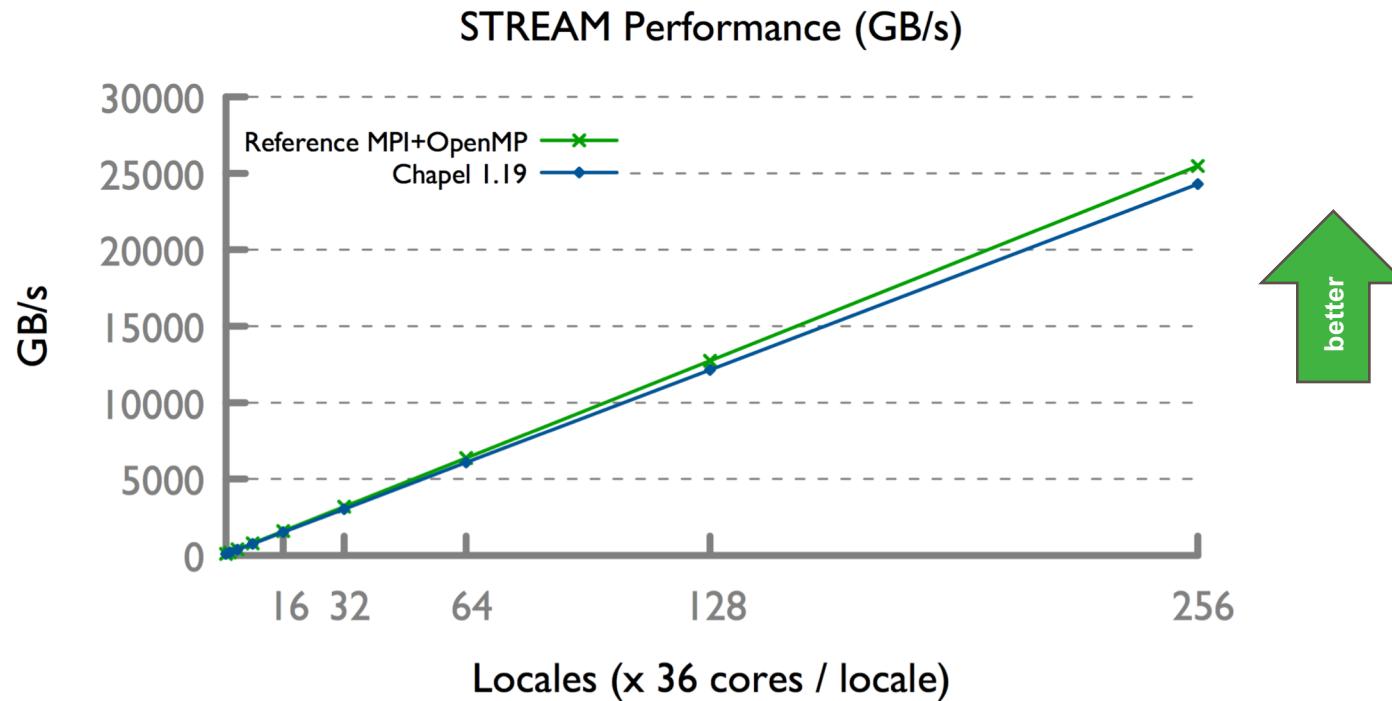
```
int HPCC_Stream(HPCC_Parms *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HP
```

```
a = HPCC_XMALLOC();
b = HPCC_XMALLOC();
c = HPCC_XMALLOC();
```



HPCC STREAM Triad: Chapel vs. C+MPI+OpenMP

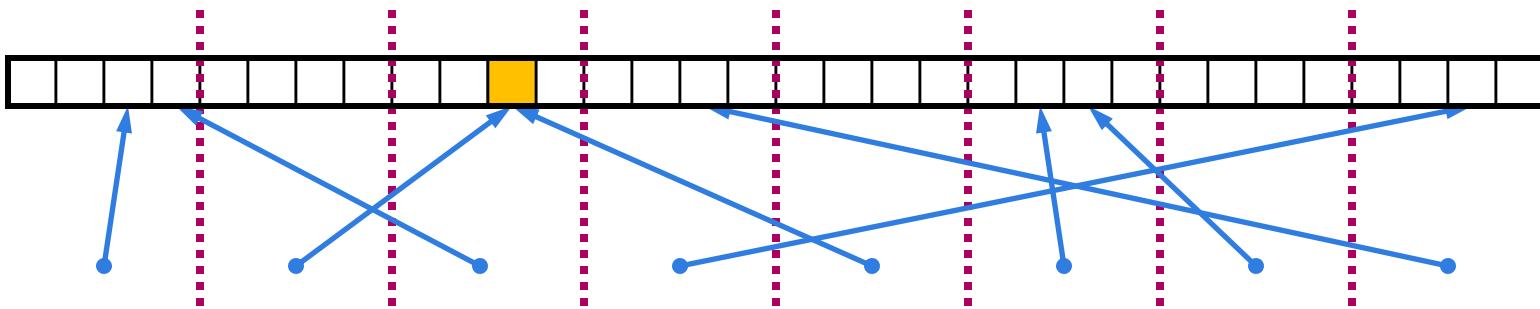
CRAY



HPCC Random Access (RA)

CRAY

Data Structure: distributed table



Computation: update random table locations in parallel

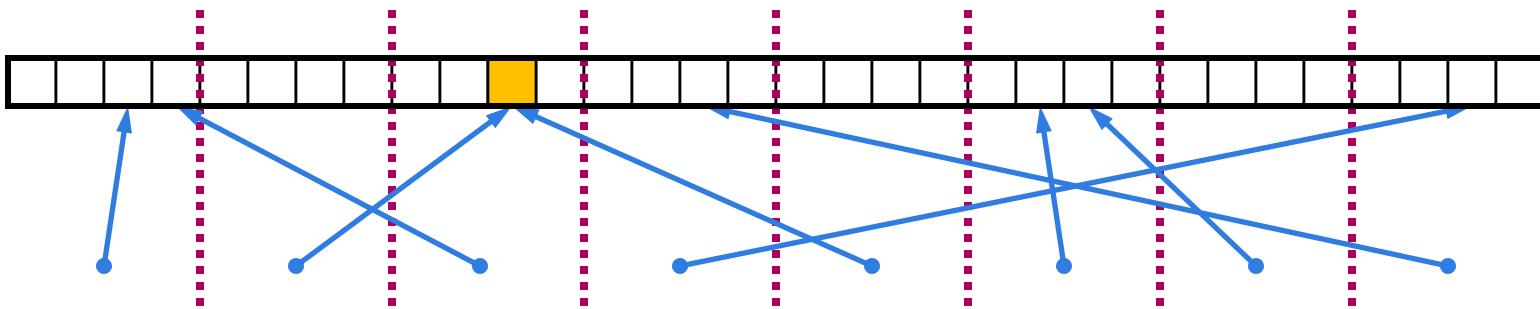
Two variations:

- **lossless:** don't allow any updates to be lost
- **lossy:** permit some fraction of updates to be lost

HPCC Random Access (RA)

CRAY

Data Structure: distributed table



Computation: update random table locations in parallel

Two variations:

- ➡ • **lossless:** don't allow any updates to be lost ←
- **lossy:** permit some fraction of updates to be lost

HPCC RA: MPI kernel

CRAY

```

/* Perform updates to main table. The scalar equivalent is:
 *
 * for (i=0; i<NUPDATE; i++) {
 *   Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
 *   Table[Ran & (TABSIZE-1)] ^= Ran;
 * }
 */

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
while (i < SendCnt) {
    /* receive messages */
    do {
        MPI_Test(&inreq, &have_done, &status);
        if (have_done) {
            if (status.MPI_TAG == UPDATE_TAG) {
                MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
                bufferBase = 0;
                for (j=0; j < recvUpdates; j++) {
                    inmsg = LocalRecvBuffer[bufferBase+j];
                    LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                                  tparams.GlobalStartMyProc;
                    HPCC_Table[LocalOffset] ^= inmsg;
                }
            } else if (status.MPI_TAG == FINISHED_TAG) {
                NumberReceiving--;
            } else
                MPI_Abort( MPI_COMM_WORLD, -1 );
            MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
        }
    } while (have_done && NumberReceiving > 0);
    if (pendingUpdates < maxPendingUpdates) {
        Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
        GlobalOffset = Ran & (tparams.TableSize-1);
        if (GlobalOffset < tparams.Top)
            WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) );
        else
            WhichPe = ( (GlobalOffset - tparams.Remainder) /
                        tparams.MinLocalTableSize );
        if (WhichPe == tparams.MyProc) {
            LocalOffset = (Ran & (tparams.TableSize - 1)) -
                          tparams.GlobalStartMyProc;
            HPCC_Table[LocalOffset] ^= Ran;
        }
    }
}

    } else {
        HPCC_InsertUpdate(Ran, WhichPe, Buckets);
        pendingUpdates++;
    }
    i++;
}
else {
    MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
    if (have_done) {
        outreq = MPI_REQUEST_NUL;
        pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                             &peUpdates);
        MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
                  UPDATE_TAG, MPI_COMM_WORLD, &outreq);
        pendingUpdates -= peUpdates;
    }
}
/* send remaining updates in buckets */
while (pendingUpdates > 0) {
    /* receive messages */
    do {
        MPI_Test(&inreq, &have_done, &status);
        if (have_done) {
            if (status.MPI_TAG == UPDATE_TAG) {
                MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
                bufferBase = 0;
                for (j=0; j < recvUpdates; j++) {
                    inmsg = LocalRecvBuffer[bufferBase+j];
                    LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                                  tparams.GlobalStartMyProc;
                    HPCC_Table[LocalOffset] ^= inmsg;
                }
            } else if (status.MPI_TAG == FINISHED_TAG) {
                /* we got a done message. Thanks for playing... */
                NumberReceiving--;
            } else {
                MPI_Abort( MPI_COMM_WORLD, -1 );
            }
            MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
        }
    } while (have_done && NumberReceiving > 0);
}

MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
if (have_done) {
    outreq = MPI_REQUEST_NUL;
    pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                         &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
              UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
}
/* send our done messages */
for (proc_count = 0 ; proc_count < tparams.NumProcs ; ++proc_count) {
    if (proc_count == tparams.MyProc) { tparams.finish_req(tparams.MyProc) =
                                         MPI_REQUEST_NUL; continue; }
    /* send garbage - who cares, no one will look at it */
    MPI_Isend(&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG,
              MPI_COMM_WORLD, tparams.finish_req + proc_count);
}
/* Finish everyone else up... */
while (NumberReceiving > 0) {
    MPI_Wait(&inreq, &status);
    if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j++) {
            inmsg = LocalRecvBuffer[bufferBase+j];
            LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                          tparams.GlobalStartMyProc;
            HPCC_Table[LocalOffset] ^= inmsg;
        }
    } else if (status.MPI_TAG == FINISHED_TAG) {
        /* we got a done message. Thanks for playing... */
        NumberReceiving--;
    } else {
        MPI_Abort( MPI_COMM_WORLD, -1 );
    }
    MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
              MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
}
MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses);

```



HPCC RA: MPI kernel comment vs. Chapel

CRAY

```

/* Perform updates to main table. The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
 *     Table[Ran & (TABSIZEx1)]^=Ran;
 *   }
 */

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD);
while (i < SendCount) {
    /* receive messages */
    do {
        MPI_Test(&inreq, &have_done, &status);
        if (have_done) {
            if (status.MPI_TAG == UPDATE_TAG) {
                MPI_Get_count(&status, tparams.dtype64,
                             bufferBase = 0;

```

Chapel Kernel

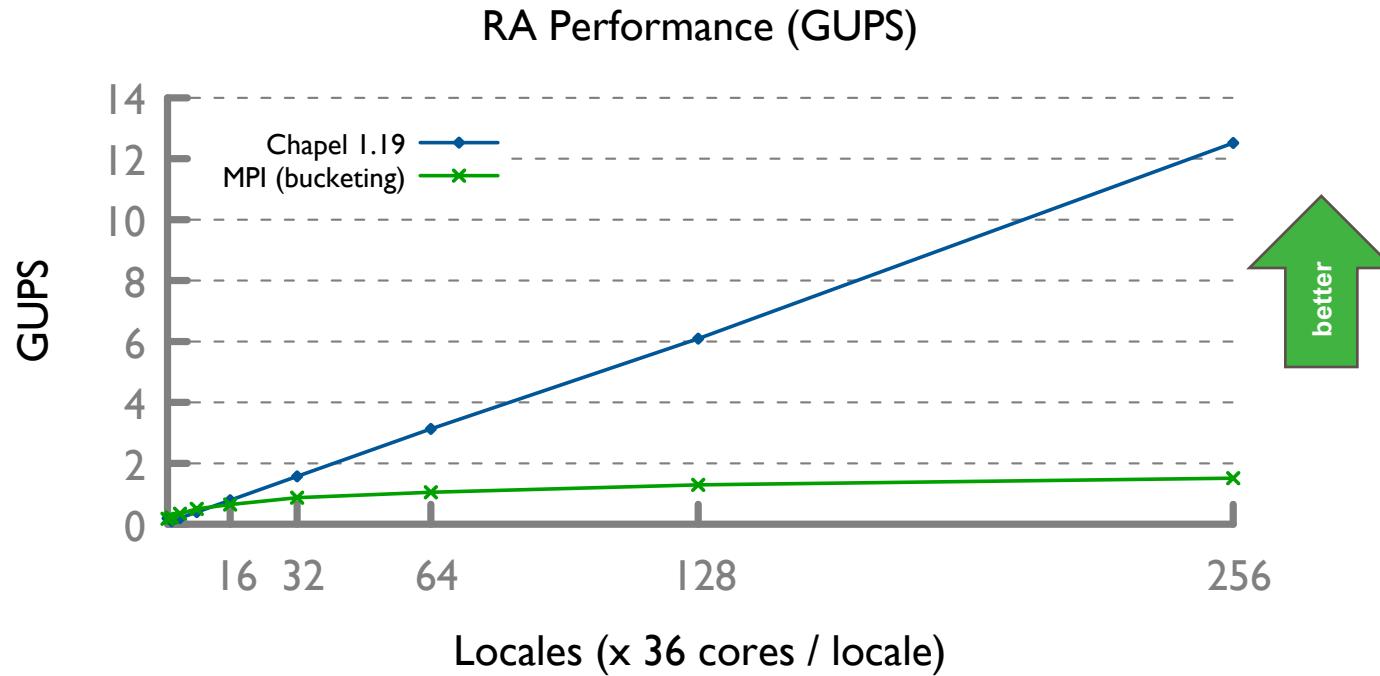
```
forall (_ , r) in zip(Updates, RASTream()) do
    T[r & indexMask].xor(r);
```

MPI Comment

```
/* Perform updates to main table. The scalar equivalent is:  
*  
*      for (i=0; i<NUPDATE; i++) {  
*          Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);  
*          Table[Ran & (TABSIZ-1)] ^= Ran;  
*      }  
*/
```

HPCC RA: Chapel vs. C+MPI

CRAY



HPCC RA: MPI vs. Chapel

CRAY

/ Perform updates to main table. The scalar equivalent is:*

```
*      for (i=0; i<NUPDATE; i++) {
*          Ran = (Ran << 1) ^ ((s64int) Ran < 0) ? POLY : 0;
*          Table[Ran & (TABSIZE-1)]^= Ran;
*      }
*
MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
          while (i < SendCnt) {
    /* receive messages */
    do {
        MPI_Test(&inreq, &have_done, &status);
        if (have_done) {
            if (status.MPI_TAG == UPDATE_TAG) {
                MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
                bufferBase = 0;
                for (j=0; j < recvUpdates; j++) {
                    inmsg = LocalRecvBuffer[bufferBase+j];
                    LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                                  tparams.GlobalStartMyProc;
                    HPCC_Table[LocalOffset] ^= inmsg;
                }
            } else if (status.MPI_TAG == FINISHED_TAG) {
                NumberReceiving--;
            } else {
                MPI_Abort( MPI_COMM_WORLD, -1 );
                MPI_Irecv(&LocalRecvBuffer, localbufferSize, tparams.dtype64,
                          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
            }
        }
    } while (have_done && NumberReceiving > 0);
    if (pendingUpdates < maxPendingUpdates) {
        Ran = (Ran << 1) ^ ((s64int) Ran < ZERO64B ? POLY : ZERO64B);
        GlobalOffset = Ran & (tparams.TableSize-1);
        if (GlobalOffset < tparams.Top)
            WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) ) % tparams.PE;
        else
            WhichPe = ( (GlobalOffset - tparams.Remainder) /
                        tparams.MinLocalTableSize );
        if (WhichPe == tparams.MyProc) {
            LocalOffset = (Ran & (tparams.TableSize - 1)) -
                          tparams.GlobalStartMyProc;
            HPCC_Table[LocalOffset] ^= Ran;
```

Chapel Kernel

```
forall (_, r) in zip(Updates, RASTream()) do
    T[r & indexMask].xor(r);
```

```
/* our done messages */
for (proc_count = 0 ; proc_count < tparams.NumProcs; ++proc_count) {
    if (proc_count == tparams.MyProc) { tparams.finish_req(tparams.MyProc) = MPI_REQUEST_NULL; continue; }
    /* send garbage - who cares, no one will look at it */
    MPI_Isend(&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG,
              MPI_COMM_WORLD, tparams.finish_req + proc_count);

/* from everyone else up... */
while (NumberReceiving > 0) {
    MPI_Wait(&inreq, &status);
    if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j++) {
            inmsg = LocalRecvBuffer[bufferBase+j];
            LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                          tparams.GlobalStartMyProc;
            HPCC_Table[LocalOffset] ^= inmsg;
        }
    } else if (status.MPI_TAG == FINISHED_TAG) {
        /* we got a done message. Thanks for playing... */
        NumberReceiving--;
    } else {
        MPI_Abort( MPI_COMM_WORLD, -1 );
    }
}
MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses );
```



HPCC RA: MPI vs. Chapel

CRAY

```
/* Perform updates to main table. The scalar equivalent is:
 *
 * for (i=0; i<NUPDATE; i++) {
 *   Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
 *   Table[Ran & (TABSIZE-1)] ^= Ran;
 * }
 *
MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
while (i < SendCnt) {
  /* receive messages */
  do {
    MPI_Test(&inreq, &have_done, &status);
    if (have_done) {
      if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j++) {
          inmsg = LocalRecvBuffer[bufferBase+j];
          LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                        tparams.GlobalStartMyProc;
          HPCC_Table[LocalOffset] ^= inmsg;
        }
      } else if (status.MPI_TAG == FINISHED_TAG) {
        NumberReceiving--;
      } else
        MPI_Abort( MPI_COMM_WORLD, -1 );
      MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
    }
  } while (have_done && NumberReceiving > 0);
  if (pendingUpdates < maxPendingUpdates) {
    Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
    GlobalOffset = Ran & (tparams.TableSize-1);
    if (GlobalOffset < tparams.Top)
      WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) );
    else
      WhichPe = ( (GlobalOffset - tparams.Remainder) /
                  tparams.MinLocalTableSize );
    if (WhichPe == tparams.MyProc) {
      LocalOffset = (Ran & (tparams.TableSize - 1)) -
                    tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= Ran;
    }
    HPCC_InsertUpdate(Ran, WhichPe, Buckets);
  }
}
```

Chapel Kernel

```
forall (_, r) in zip(Updates, RASTream()) do
  T[r & indexMask].xor(r);
```

```
      pendingUpdates);
      MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
                UPDATE_TAG, MPI_COMM_WORLD, &outreq);
      pendingUpdates -= peUpdates;
    }
  }
  /* send remaining updates in buckets */
  while (pendingUpdates > 0) {
    /* receive messages */
    do {
      MPI_Test(&inreq, &have_done, &status);
      if (have_done) {
        if (status.MPI_TAG == UPDATE_TAG) {
          MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
          bufferBase = 0;
          for (j=0; j < recvUpdates; j++) {
            inmsg = LocalRecvBuffer[bufferBase+j];
            LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                          tparams.GlobalStartMyProc;
            HPCC_Table[LocalOffset] ^= inmsg;
          }
        } else if (status.MPI_TAG == FINISHED_TAG) {
          /* we got a done message. Thanks for playing... */
          NumberReceiving--;
        } else {
          MPI_Abort( MPI_COMM_WORLD, -1 );
        }
        MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                  MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
      }
    } while (have_done && NumberReceiving > 0);
  }
  /* send our done messages */
  for (proc_count = 0; proc_count < tparams.NumProcs ; ++proc_count) {
    if (proc_count == tparams.MyProc) { tparams.finish_req(tparams.MyProc) =
      MPI_REQUEST_NULL; continue; }
    /* send garbage - who cares, no one will look at it */
    MPI_Isend(&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG,
              MPI_COMM_WORLD, tparams.finish_req + proc_count);
  }
  /* Finish everyone else up... */
  while (NumberReceiving > 0) {
    MPI_Wait(&inreq, &status);
    if (status.MPI_TAG == UPDATE_TAG) {
      MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
      bufferBase = 0;
      for (j=0; j < recvUpdates; j++) {
        inmsg = LocalRecvBuffer[bufferBase+j];
        LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                      tparams.GlobalStartMyProc;
        HPCC_Table[LocalOffset] ^= inmsg;
      }
    } else if (status.MPI_TAG == FINISHED_TAG) {
      /* we got a done message. Thanks for playing... */
      NumberReceiving--;
    } else {
      MPI_Abort( MPI_COMM_WORLD, -1 );
    }
    MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
              MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
  }
  MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses);
```



Why Consider New Languages at all?

CRAY

Syntax

- High level, elegant syntax
- Improve programmer productivity

Semantics

- Static analysis can help with correctness
- We need a compiler (front-end)

Performance

- If optimizations are needed to get performance
- We need a compiler (back-end)

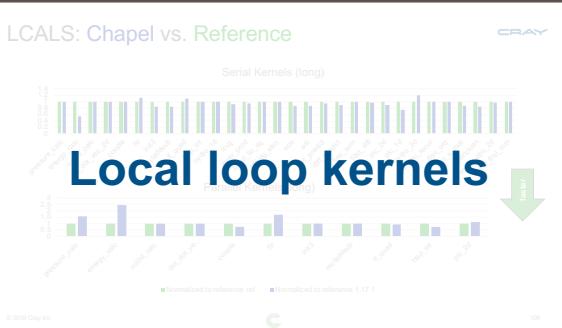
Algorithms

- Language defines what is easy and hard
- Influences algorithmic thinking

[Source: Kathy Yelick,
CHI UW 2018 keynote:
*Why Languages Matter
More Than Ever*]

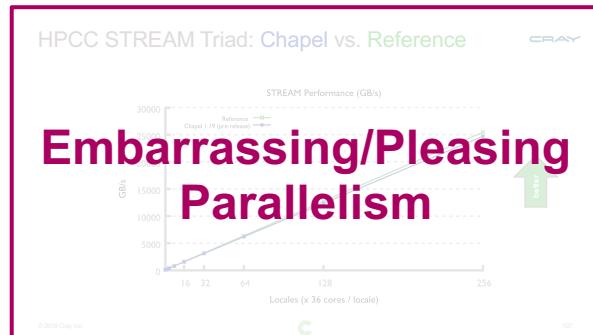
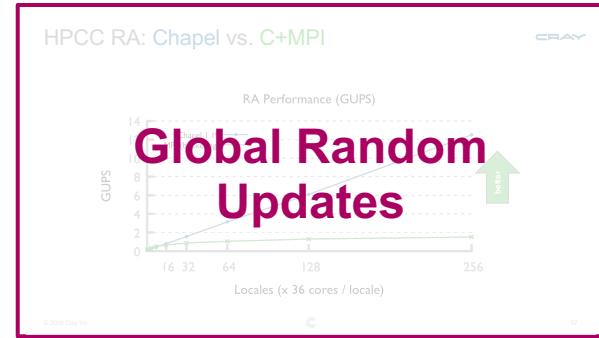
HPC Patterns: Chapel vs. Reference

CRAY



LCALS

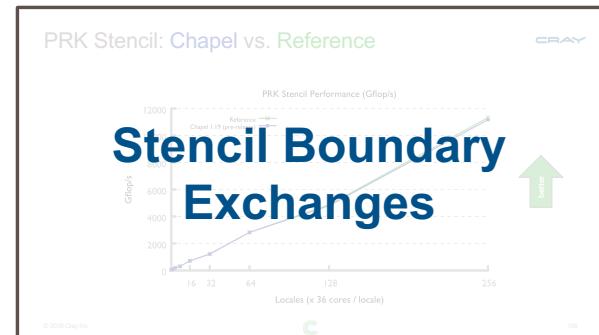
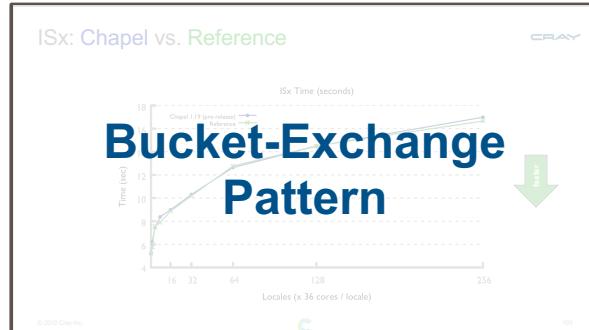
HPCC RA



STREAM
Triad

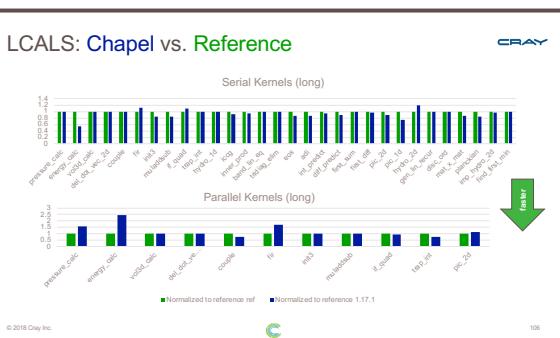
ISx

PRK
Stencil



HPC Patterns: Chapel vs. Reference

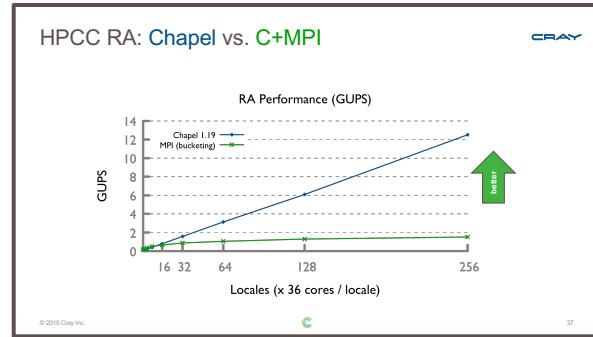
CRAY



LCALS



HPCC RA



HPCC STREAM Triad: Chapel vs. Reference

STREAM Performance (GB/s)

Locales	Chapel 1.19 (pre-release) (GB/s)	Reference (GB/s)
0	0	0
16	~2000	~2000
32	~4000	~4000
64	~6000	~6000
128	~12000	~12000
256	~25000	~25000

GB/s

Locales (x 36 cores / locale)

Chapel 1.19 (pre-release)

Reference

better

iSx: Chapel vs. Reference

Locales (x 36 cores / locale)	Chapel 1.19 (pre-optimized) (sec)	Reference (sec)
4	~4.5	~6.5
16	~8.5	~9.5
32	~9.5	~10.5
64	~12.5	~13.5
128	~14.5	~15.5
256	~16.5	~17.5

ISX Time (seconds)

Time (sec)

Locales (x 36 cores / locale)

Chapel 1.19 (pre-optimized)

Reference

ISX

PRK Stencil: Chapel vs. Reference

PRK Stencil Performance (Gflop/s)

Locales (x 36 cores / locale)	Chapel 1.9 (pre-release) Gflop/s	Reference Gflop/s
1	~100	~100
16	~2500	~3000
32	~4500	~5500
64	~7500	~9000
128	~12000	~14000
256	~15000	~18000

A green arrow pointing upwards indicates that the Reference implementation is better than the Chapel 1.9 (pre-release) implementation.

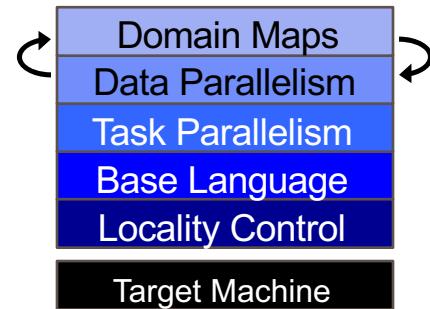
A Brief Tour of Chapel Features



Chapel Feature Areas

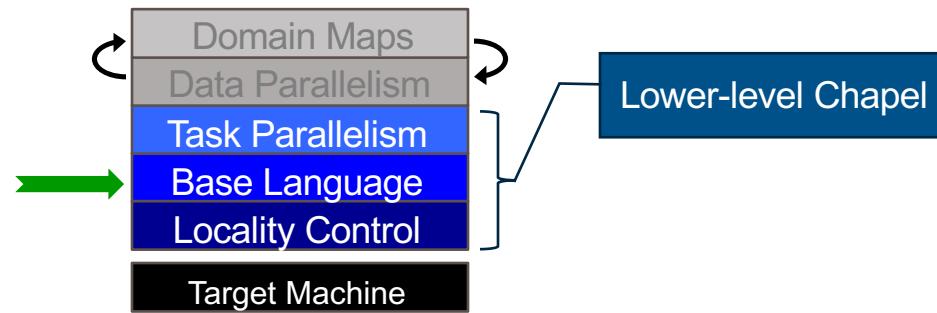
CRAY

Chapel language concepts



Base Language

CRAY



Base Language Features, by example

CRAY

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

```
config const n = 10;  
  
for f in fib(n) do  
    writeln(f);
```

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

Base Language Features, by example

CRAY

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

```
config const n = 10;  
  
for f in fib(n) do  
    writeln(f);
```

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

Configuration declarations
(support command-line overrides)
.fib --n=1000000

Base Language Features, by example

CRAY

Iterators

```
iter fib(n) {  
    var current = 0,  
        next = 1;  
  
    for i in 1..n {  
        yield current;  
        current += next;  
        current <=gt; next;  
    }  
}
```

```
config const n = 10;  
  
for f in fib(n) do  
    writeln(f);
```

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

Base Language Features, by example

Static type inference for:

- arguments
- return types
- variables

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

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

```
config const n = 10;

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

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

Base Language Features, by example

CRAY

Explicit types also supported

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

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

```
config const n: int = 10;

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

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

Base Language Features, by example

CRAY

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

```
config const n = 10;  
  
for f in fib(n) do  
    writeln(f);
```

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

Base Language Features, by example

CRAY

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

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

```
config const n = 10;

for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
...
```

Zippered iteration

Base Language Features, by example

CRAY

Range types and operators

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

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

```
config const n = 10;

for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
...
```

Base Language Features, by example

CRAY

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

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

```
config const n = 10;

for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

Tuples

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
...
```

Base Language Features, by example

CRAY

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

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

```
config const n = 10;

for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
...
```

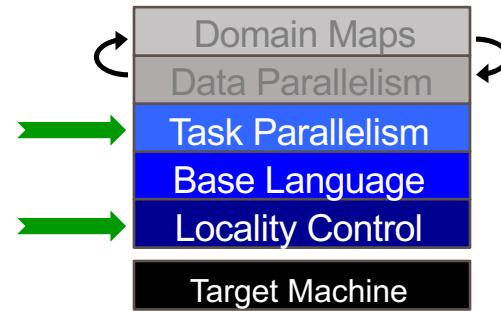
Other Base Language Features

CRAY

- **Object-oriented programming** (value- and reference-based)
 - Managed objects and lifetime checking
 - Nilable vs. non-nilable class variables
- **Generic programming / polymorphism**
- **Error-handling**
- **Compile-time meta-programming**
- **Modules** (supporting namespaces)
- **Procedure overloading / filtering**
- **Arguments:** default values, intents, name-based matching, type queries
- and more...

Task Parallelism and Locality Control

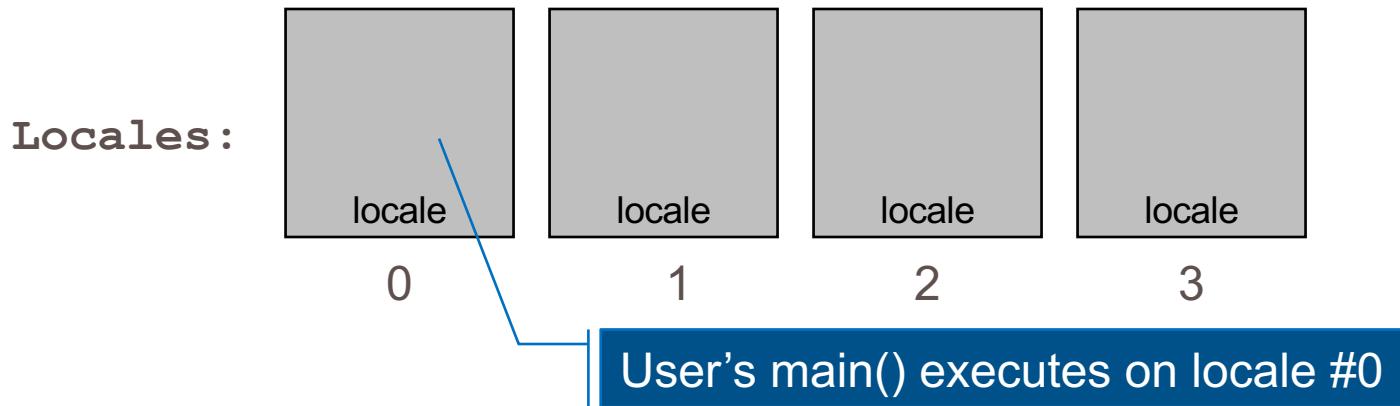
CRAY



Locales, briefly

- Locales can run tasks and store variables
 - Think “compute node”
 - Number of locales specified on execution command-line

```
> ./myProgram --numLocales=4      # or ` -nl 4`
```



Task Parallelism and Locality, by example

CRAY

taskParallel.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writef("Hello from task %n of %n "+
           "running on %s\n",
           tid, numTasks, here.name);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

Abstraction of
System Resources

taskParallel.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writef("Hello from task %n of %n "+
           "running on %s\n",
           tid, numTasks, here.name);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

High-Level
Task Parallelism

taskParallel.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writef("Hello from task %n of %n "+
           "running on %s\n",
           tid, numTasks, here.name);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

So far, this is a shared memory program
Nothing refers to remote locales,
explicitly or implicitly

taskParallel.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writef("Hello from task %n of %n "+
           "running on %s\n",
           tid, numTasks, here.name);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

taskParallel.chpl

```
coforall loc in Locales do
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            writef("Hello from task %n of %n "+
                "running on %s\n",
                tid, numTasks, here.name);
    }
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

Abstraction of
System Resources

taskParallel.chpl

```
coforall loc in Locales do
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            writef("Hello from task %n of %n "+
                "running on %s\n",
                tid, numTasks, here.name);
    }
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

High-Level
Task Parallelism

taskParallel.chpl

```
coforall loc in Locales do
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            writef("Hello from task %n of %n "+
                "running on %s\n",
                tid, numTasks, here.name);
    }
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

Control of Locality/Affinity

taskParallel.chpl

```
coforall loc in Locales do
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            writef("Hello from task %n of %n "+
                "running on %s\n",
                tid, numTasks, here.name);
    }
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

taskParallel.chpl

```
coforall loc in Locales do
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            writef("Hello from task %n of %n "+
                "running on %s\n",
                tid, numTasks, here.name);
    }
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

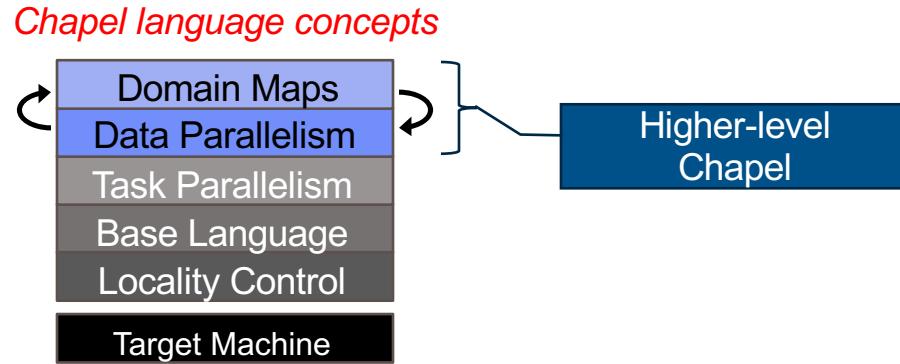
Other Task Parallel Features

CRAY

- **atomic / synchronized variables:** for sharing data & coordination
- **begin / cobegin statements:** other ways of creating tasks

Data Parallelism in Chapel

CRAY



Data Parallelism, by example

CRAY

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Data Parallelism, by example

CRAY

Domains (Index Sets)

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Data Parallelism, by example

CRAY

Arrays

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Data Parallelism, by example

CRAY

Data-Parallel Forall Loops

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Data Parallelism, by example

CRAY

So far, this is a shared memory program
Nothing refers to remote locales,
explicitly or implicitly

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Distributed Data Parallelism, by example

CRAY

Domain Maps
(Map Data Parallelism to the System)

dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Distributed Data Parallelism, by example

CRAY

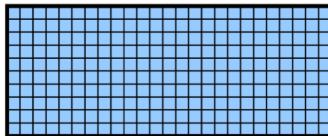
dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

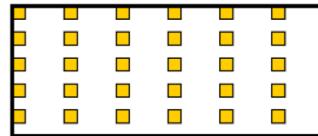
```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Other Data Parallel Features

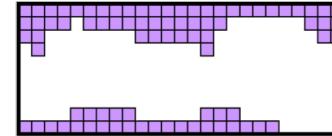
- **Parallel Iterators and Zippering**
- **Slicing:** refer to subarrays using ranges / domains
- **Promotion:** execute scalar functions in parallel using array arguments
- **Reductions:** collapse arrays to scalars or subarrays
- **Scans:** parallel prefix operations
- **Several Domain/Array Types**



dense



strided



sparse



associative

STREAM Triad and HPCC RA Kernel, revisited

CRAY

```
use ...;

config const m = 1000,
          alpha = 3.0;

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

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

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

```
forall (_, r) in zip(Updates, RAStruct()) do
    T[r & indexMask].xor(r);
```

Arkouda: NumPy over Chapel



Arkouda: Context

CRAY

Motivation: Say you've got...

- ...a bunch of Python programmers
- ...HPC-scale problems to solve
- ...access to HPC systems

How should you leverage these Python programmers to get your work done?

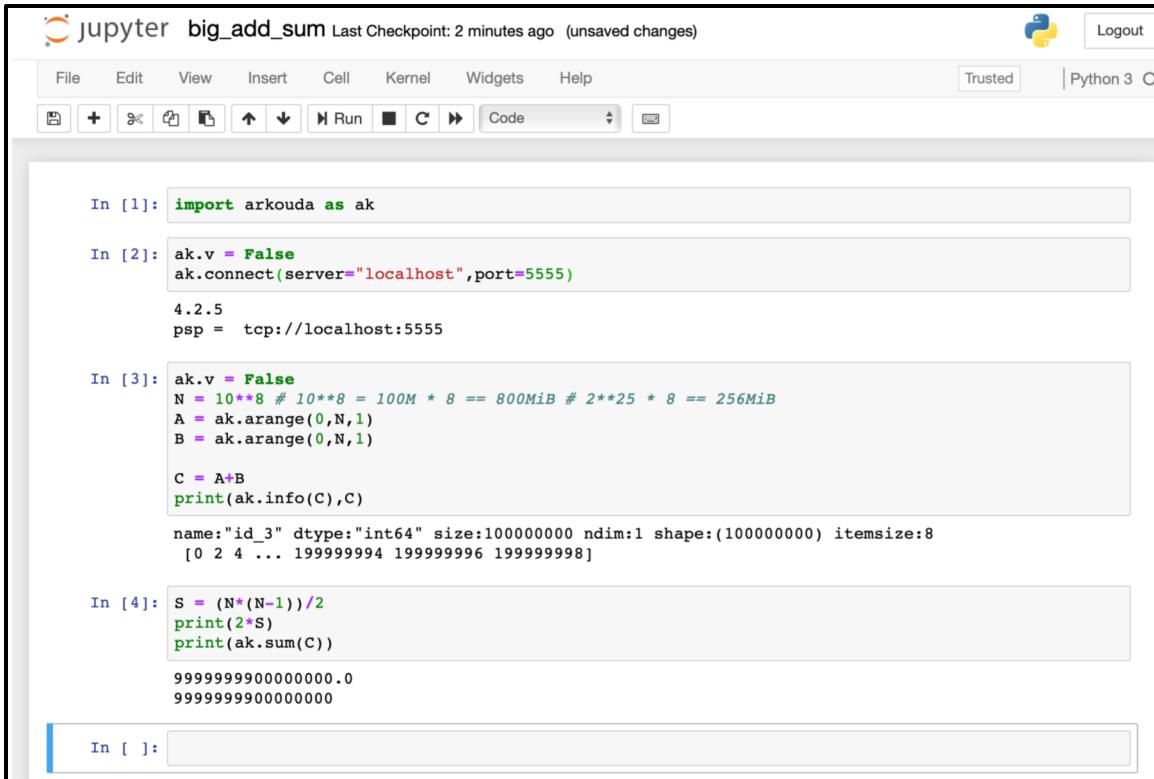
Concept: Develop Python libraries that are implemented in Chapel

⇒ get performance, as with Python-over-C, but also parallelism + scalability

Even Better: use NumPy interfaces to make it trivial / familiar for users

Sample use from Jupyter

CRAY



The screenshot shows a Jupyter Notebook interface with the following code and output:

```
In [1]: import arkouda as ak

In [2]: ak.v = False
ak.connect(server="localhost", port=5555)

4.2.5
psp = tcp://localhost:5555

In [3]: ak.v = False
N = 10**8 # 10**8 = 100M * 8 == 800MiB # 2**25 * 8 == 256MiB
A = ak.arange(0,N,1)
B = ak.arange(0,N,1)

C = A+B
print(ak.info(C),C)

name:"id_3" dtype:"int64" size:100000000 ndim:1 shape:(100000000) itemsize:8
[0 2 4 ... 199999994 199999996 199999998]

In [4]: S = (N*(N-1))/2
print(2*S)
print(ak.sum(C))

999999900000000.0
9999999000000000

In [ ]:
```

Arkouda Accomplishments

CRAY

By taking this approach, this user was able to:

- interact with a distributed, running Chapel program from Python within Jupyter
- run the same back-end program on...
 - ...a Cray XC
 - ...an HPE Superdome X
 - ...an Infiniband cluster
 - ...a Mac laptop
- compute on TB-sized arrays in seconds
- with ~1 month of effort

“Why Chapel?”

CRAY

- **High level — makes for less code**
 - Great support for array operations and distributed arrays
 - Direct support for synchronized/atomic variables
 - Close to “Pythonic” (for a statically typed language)
 - Provides a gateway for data scientists ready to go beyond Python
- **Portability and Scalability**
 - Same code runs on shared- or distributed-memory systems
 - “From Raspberry Pi to Cray XC”
- **Integrates with (distributed) numerical libraries** (e.g., FFTW, FFTW-MPI)

Summary and Resources



Summary of this Talk

Chapel cleanly and orthogonally supports...

...expression of parallelism and locality

...specifying how to map computations to the system

Chapel is powerful:

- supports succinct, straightforward code
- can result in performance that competes with (or beats) C+MPI+OpenMP

Chapel is attractive to Python programmers

- as a native language: similarly readable / writeable, yet scalable
- as an implementation option for Python libraries

Chapel Central

CRAY

<https://chapel-lang.org>

- downloads
- presentations
- papers
- resources
- documentation



The Chapel Parallel Programming Language

What is Chapel?

Chapel is a modern programming language that is...

- **parallel:** contains first-class concepts for concurrent and parallel computation
- **productive:** designed with programmability and performance in mind
- **portable:** runs on laptops, clusters, the cloud, and HPC systems
- **scalable:** supports locality-oriented features for distributed memory systems
- **open-source:** hosted on [GitHub](#), permissively [licensed](#)

New to Chapel?

As an introduction to Chapel, you may want to...

- read a [blog article](#) or [book chapter](#)
- watch an [overview talk](#) or browse its [slides](#)
- [download](#) the release
- browse [sample programs](#)
- view [other resources](#) to learn how to trivially write distributed programs like this:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;      // use --n=<val> when executing to override this default
forall i in {1..n} dmapped Cyclic(startIdx=1) do
    writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```

What's Hot?

- **Chapel 1.17** is now available—[download](#) a copy or browse its [release notes](#)
- The [advance program](#) for **CHI UW 2018** is now available—hope to see you there!
- Chapel is proud to be a [Rails Girls Summer of Code 2018 organization](#)
- Watch talks from [ACCU 2017](#), [CHI UW 2017](#), and [ATPESC 2016](#) on [YouTube](#)
- [Browse slides](#) from **SIAM PP18**, **NWCPP**, **SeaLang**, **SC17**, and other recent talks
- Also see: [What's New?](#)



Chapel Online Documentation

CRAY

<https://chapel-lang.org/docs>: ~200 pages, including primer examples

The screenshot displays the Chapel Online Documentation website, version 1.17. The main navigation bar includes links for "Docs", "Chapel Documentation", "version 1.17 ▾", "Search docs", and "View page source". The sidebar contains links for "COMPILING AND RUNNING CHAPEL" (Quickstart Instructions, Using Chapel, Platform-Specific Notes, Technical Notes, Tools) and "WRITING CHAPEL PROGRAMS" (Quick Reference, Hello World Variants, Primers, Language Specification, Built-in Types and Functions, Standard Modules, Package Modules, Standard Layouts and Distributions, Chapel Users Guide (WIP)). The main content area shows the "Chapel Documentation" page, which includes sections for "Compiling and Running Chapel" (with a list of sub-topics like Quickstart Instructions, Using Chapel, etc.) and "Writing Chapel Programs" (with a list of sub-topics like Quick Reference, Hello World Variants, Primers, Language Specification, etc.). Below these are "Language History" (Chapel Evolution, Archived Language Specifications) and "Platform-Specific Notes" (Technical Notes). A "View page source" link is located at the top right of the main content area. To the right, there are three smaller examples of documentation pages: "Using Chapel", "Task Parallelism", and "Cobegin Statements". Each example shows a similar header and sidebar structure.



Chapel Community

CRAY

Questions Developer Jobs Tags Users [chapel]

Tagged Questions

Chapel is a portable, open-source parallel programming language. Use this tag to ask questions about the Chapel language or its implementation.

Learn more... Improve tag info Top users Synonyms

6 votes 1 answer 79 views

6 votes 1 answer 47 views

6 votes 2 answers 47 views

<https://stackoverflow.com/questions/tagged/chapel>

This repository Search Pull requests Issues Marketplace Gist

chapel-lang / chapel

Code Issues 292 Pull requests 26 Projects 0 Settings Insights

Filters ▾ Is:issue is:open Labels Milestones

292 Open 77 Closed

Implement "bounded-coforall" optimization for remote coforalls area: Compiler type: Performance #6357 opened 13 hours ago by ronawho

Consider using processor atomics for remote coforalls EndCount area: Compiler type: Performance #6356 opened 13 hours ago by ronawho 0 of 6

make uninstall area: BTR type: Feature Request #6353 opened 14 hours ago by mpf

make check doesn't work with ./configure area: BTR #6352 opened 16 hours ago by mpf

Passing variable via intent to a forall loop seems to create an iteration-private variable, not a task-private one area: Compiler type: Bug #6351 opened a day ago by casselle

Remove chpl_comm_make_progress area: Runtime easy type: Design #6349 opened a day ago by sunghunchoi

Runtime error after make on Linux Mint area: BTR user issue #6348 opened a day ago by denindiana

<https://github.com/chapel-lang/chapel/issues>

GITTER

chapel-lang/chapel Chapel programming language | Peak developer hours are 0600-1700 PT

Brian Dolan @buddha314 what's the syntax for making a copy (not a reference) to an array? May 09 14:34

Michael Ferguson @mpff like in a new variable? May 09 14:40

```
var A[1..10] int;
var B = A; // makes a copy of A
ref C = A; // refers to A
```

Brian Dolan @buddha314 oh, got it, thanks! May 09 14:41

Michael Ferguson @mpff May 09 14:42

```
proc f(x) { /* x refers to the actual argument */ }
proc g(in arr) { /* arr is a copy of the actual argument */ }
var A[1..10] int;
f(A);
g(A);
```

Brian Dolan @buddha314 isn't there a proc f(ref arr) {} as well? May 09 14:43

Michael Ferguson @mpff yes. The default intent for array is 'ref' or 'const ref' depending on if the function body modifies it. So that's effectively the default. May 09 14:55

Brian Dolan @buddha314 thanks! May 09 14:55

<https://gitter.im/chapel-lang/chapel>

read-only mailing list: chapel-announce@lists.sourceforge.net (~15 mails / year)

Chapel Social Media (no account required)

CRAY

[http://twitter.com/ChapelLanguage](https://twitter.com/ChapelLanguage)

[http://facebook.com/ChapelLanguage](https://facebook.com/ChapelLanguage)

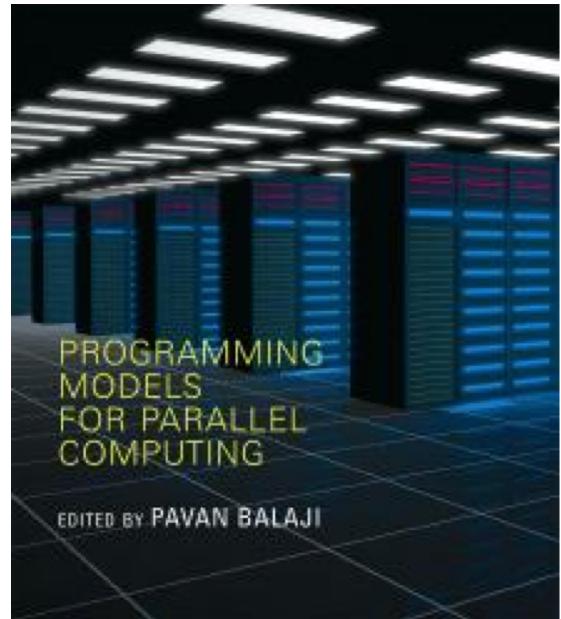
<https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/>

Suggested Reading: Chapel history and overview

CRAY

Chapel chapter from *[Programming Models for Parallel Computing](#)*

- a detailed overview of Chapel's history, motivating themes, features
- published by MIT Press, November 2015
- edited by Pavan Balaji (Argonne)
- chapter is also available [online](#)



Suggested Reading: Recent Progress (CUG 2018)

Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson,
Ben Hershberger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus
Chapel Team
Cray Inc.
Seattle, WA, USA
chapel_info@cray.com

Abstract—Chapel is a programming language whose goal is to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as combining the strengths of Python, Fortran, C/C++, and MPI in a single language. Over years, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel wrapped up, and the team embarked on a five-year effort to move the Chapel compiler to end-users. This paper follows up on our CUG 2016 paper summarizing the progress made by the Chapel project since that time. Specifically, Chapel's performance now competes with or beats hand-coded SIMD/FIESTA, the suite of libraries ZMQ has grown to include PETSc, Raja, LAMMPS, MPI, and other key technologies; its documentation has been modernized and fleshed out; and the set of tools available to Chapel users has grown. This paper also characterizes the experiences of contributors from communities as diverse as astrophysics and artificial intelligence.

Keywords—Parallel programming; Computer languages

I. INTRODUCTION

Chapel is a programming language designed to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as striving to create a language whose code is as attractive to read and write as Python, yet which supports the performance of Fortran and the scalability of MPI. Chapel also aims to compete with C in terms of portability, and with C++ in terms of flexibility and extensibility. Chapel is designed to be general-purpose in the sense that when you have a parallel algorithm in mind and want to specify exactly how to run it, Chapel should be able to handle that scenario.

Chapel's design and implementation are led by Cray Inc., with feedback and code contributed by users and the open-source community. Though developed by Cray, Chapel's design and implementation are portable, permitting its programs to scale up from multicore laptops to commodity clusters to Cray systems. In addition, Chapel programs can be run on cloud-computing platforms and HPC systems from other vendors. Chapel is being developed in an open-source manner under the Apache 2.0 license and is hosted at GitHub.¹

¹<https://github.com/chapel-lang/chapel>

paper and slides available at chapel-lang.org



The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems program (HPCS). HPCS wrapped up in late 2012, at which point Chapel was a compelling prototype, having successfully demonstrated several key research challenges that the project had undertaken. Chief among these was supporting data- and task-parallelism in a single unified language, the Chapel language. This was accomplished by supporting the creation of high-level data-parallel abstractions such as parallel loops and arrays in terms of lower-level Chapel features such as classes, iterators, and tasks.

Under HPCS, Chapel also successfully supported the expression of parallelism using distinct language features from those used to control locality and affinity—that is, Chapel programmers specify which computations should run in parallel and from specifying where those computations should be run. This allows Chapel programs to support multicores, multi-node, and heterogeneous computing within a single unified language.

Chapel's implementation under HPCS demonstrated that the language could be implemented portably while still being optimized for HPC-specific features such as the RDMA support available in Cray® Gemini™ and Aries™ networks. This allows Chapel to take advantage of native hardware support for remote puts, gets, and atomic memory operations.

Despite these successes, at the close of HPCS, Chapel was not at all ready to support production codes in the field. This was not surprising given the language's aggressive design and modest-size research team. However, reactions from potential users were sufficiently positive that, in early 2013, Cray embarked on a follow-up effort to improve Chapel and move it towards being a production-ready language. Colloquially, we refer to this as "the second five-year push." This paper's contribution is to describe the results of this five-year effort, providing readers with an understanding of Chapel's progress and achievements since the end of the HPCS program. In doing so, we directly compare the status of Chapel version 1.17, released last month, with Chapel version 1.7, which was released five years ago in April 2013.

**Chapel Comes of Age:
Productive Parallelism at Scale** 
CUG 2018
Brad Chamberlain, Chapel Team, Cray Inc.

The 6th Annual Chapel Implementers and Users Workshop

- sponsored by ACM SIGPLAN
- held in conjunction with PLDI 2019, FCRC 2019
- June 22-23, Phoenix AZ

Keynote: Anshu Dubey (Argonne)

Programming Abstractions for Orchestration of HPC Scientific Computing

Community Talks:

- Arkouda
- hybrid CPU-GPU computations with Chapel
- Stencil computations: getting performance for a student exercise
- Chapel Graph Library

Cray talks:

- Chapel's use in AI / HPO
- interoperability improvements for Python, C, and Fortran
- recent performance optimizations
- radix sort in Chapel

Also: state of the project, lightning talks, coding day

Summary of this Talk

Chapel cleanly and orthogonally supports...

...expression of parallelism and locality

...specifying how to map computations to the system

Chapel is powerful:

- supports succinct, straightforward code
- can result in performance that competes with (or beats) C+MPI+OpenMP

Chapel is attractive to Python programmers

- as a native language: similarly readable / writeable, yet scalable
- as an implementation option for Python libraries

SAFE HARBOR STATEMENT

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts.

These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.



THANK YOU

QUESTIONS?



bradc@cray.com



@ChapelLanguage



chapel-lang.org



cray.com



@cray_inc



linkedin.com/company/cray-inc-/

