COMPILING CHAPEL: KEYS TO MAKING PARALLEL PROGRAMMING PRODUCTIVE AT SCALE

Brad Chamberlain
PACT 2020 keynote
October 7, 2020
WHAT IS CHAPEL?

Chapel: A modern parallel programming language
  • portable & scalable
  • open-source & collaborative

Goals:
  • Support general parallel programming
  • Make parallel programming at scale far more productive
WHAT DOES “PRODUCTIVITY” MEAN TO YOU?

Recent Graduates:
“Something similar to what I used in school: Python, Matlab, Java, ...”

Seasoned HPC Programmers:
“That sugary stuff which I can’t use because I need full control to ensure good performance”

Computational Scientists:
“Something that lets me focus on my science 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 need, implemented in a language that’s attractive to recent graduates.”
WHY CREATE A NEW LANGUAGE?

• Because parallel programmers deserve better
  • the state of the art for HPC is a mish-mash of libraries, pragmas, and extensions
  • parallelism and locality are concerns that deserve first-class language features

<table>
<thead>
<tr>
<th>Why Consider New Languages at all?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>• High level, elegant syntax</td>
</tr>
<tr>
<td>• Improve programmer productivity</td>
</tr>
<tr>
<td><strong>Semantics</strong></td>
</tr>
<tr>
<td>• Static analysis can help with correctness</td>
</tr>
<tr>
<td>• We need a compiler (front-end)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
</tr>
<tr>
<td>• If optimizations are needed to get performance</td>
</tr>
<tr>
<td>• We need a compiler (back-end)</td>
</tr>
<tr>
<td><strong>Algorithms</strong></td>
</tr>
<tr>
<td>• Language defines what is easy and hard</td>
</tr>
<tr>
<td>• Influences algorithmic thinking</td>
</tr>
</tbody>
</table>

• And because existing languages don’t fit our desires...

[Image Source: Kathy Yelick’s (UC Berkeley, LBNL) CHIUW 2018 keynote: Why Languages Matter More Than Ever, used with permission]
CHAPEL GOALS, RELATIVE TO OTHER LANGUAGES

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]
This talk vs. typical Chapel talks

My typical Chapel talks:
• summarize Chapel’s features and accomplishments, with the goal of growing the community

This talk:
• will do a bit of the above…
• but also give you more insights into Chapel’s compilation and optimization

Thesis: well-designed languages can improve user productivity and help with compilation
OUTLINE

I. Chapel Context & Motivation

II. Compiling Chapel Features
   ("nuts and bolts")

III. Some Chapel Optimizations
     ("bells and whistles")

IV. Summary and Resources
KEY CHAPEL FEATURES
AND HOW WE COMPILE THEM
CHAPEL FEATURE AREAS

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target System
**Base Language**

Chapel language concepts:

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

"Lower-level" Chapel

Target System
FIBONACCI ITERATION

```chapl
config const n = 10;

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

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

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

```chpl
config const n = 10;
for f in fib(n) do
  writeln(f);
iter fib(x) {
  var current = 0,
  next = 1;
  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Drive this loop by invoking `fib(n)`

‘yield’ binds values back to the loop’s index variable
FIBONACCI ITERATION

```chpl
config const n = 10;

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

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

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

And executes the loop’s body for that value

‘yield’ binds values back to the loop’s index variable
FIBONACCI ITERATION

```
config const n = 10;

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

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

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

And executes the loop's body for that value

Then the iterator continues from where it left off

Until we fall out of it (or return)

```
prompt> chpl fib.chpl
prompt> ./fib --n=20
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
...
```
FIBONACCI ITERATION

```chapl
config const n = 10;

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

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

    for i in 1..x {
        yield current;
        current += next;
        current <=> next;
    }
}
```
IMPLEMENTING SERIAL ITERATOR LOOPS

```
config const n = 10;

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

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

Rewrite the loop by inlining the iterator

```
{ const x = n;
  var current = 0,
  next = 1;
  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Then rewrite the 'yield' by inlining the loop's body

```
{ const x = n;
  var current = 0,
  next = 1;
  for i in 1..x {
    const f = current;
    writeln(f);
    current += next;
    current <=> next;
  }
}
```

More Advanced Cases
(not covered today)
- zippered iteration
- recursive iterators
OTHER BASE LANGUAGE FEATURES

- **Object-oriented programming** (value- and reference-based)
  - Nullable vs. non-nullable class variables
  - Memory-managed objects
  - Lifetime checking
- **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

**Chapel language concepts**

<table>
<thead>
<tr>
<th>Domain Maps</th>
<th>Data Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Task Parallelism</td>
</tr>
<tr>
<td></td>
<td>Base Language</td>
</tr>
<tr>
<td></td>
<td>Locality Control</td>
</tr>
</tbody>
</table>

- **Domain Maps**
- **Data Parallelism**
- **Task Parallelism**
- **Base Language**
- **Locality Control**
- **Target System**
CHAPEL TERMINOLOGY: LOCALES

- Locales can run tasks and store variables
  - Think “compute node” on a parallel system
  - User specifies number of locales on executable’s command-line

```
prompt> ./myChapelProgram --numLocales=4 # or `-nl 4`
```

**Locales array:**

- locale 0
- locale 1
- locale 2
- locale 3

User’s code starts running on locale 0
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    printf("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
```chpl
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
```

'there' refers to the locale on which we're currently running

how many processing units (think "cores") does my locale have?

what's my locale's name?
**TASK-PARALLEL “HELLO WORLD”**

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

A `coforall` loop executes each iteration as an independent task.

```
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 3 of 4 on n1032
Hello from task 2 of 4 on n1032
```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  printf("Hello from task %n of %n on %s\n", tid, numTasks, here.name);

So far, this is a shared-memory program
Nothing refers to remote locales, explicitly or implicitly
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  printf("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

```chpl
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writeln("Hello from task %n of %n on %s\n",
        tid, numTasks, here.name);
  }
}
```
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

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

create a task per locale on which the program is running

have each task run ‘on’ its locale

then print a message per core, as before

```bash
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar -numLocales=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
Hello from task 1 of 4 on n1035
...
```
IMPLEMENTING COFORALL LOOPS

```chpl
helloTaskPar.chpl

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

Outline the loop body

```chpl
proc coforallTask(argBundle) {
  printf("Hello from task %n of %n on %s\n", argBundle.tid, argBundle.numTasks, here.name);
}
```

Rewrite the loop as a serial loop that captures outer-scope variables and creates a task in each iteration

```chpl
for tid in 1..numTasks {
  const argBundle = captureVars(...);
  create_task(code=coforallTask, args=argBundle);
}
```
IMPLEMENTING COFORALL LOOPS

```chapel
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
  }
}
```
IMPLEMENTING ON-CLAUSES

helloTaskPar.chpl

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

Outline the body of the on-clause

proc onTask(argBundle) {
  const numTasks = here.numPUs();
  coforall tid in 1..numTasks do
    writeln("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
}

Rewrite the on-clause as an active message

const argBundle = captureVars(...);
create_active_msg(locale=loc, code=onTask, args=argBundle);
onClause.chpl

config const verbose = false;
var total = 0,
    done = false;

...

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1’s contribution");
      total += computeMyContribution();
  }
}
CAPTURING VARIABLES FOR ON-CLAUSES

```
config const verbose = false;
var total = 0,
    done = false;
...

on Locales[1] {
    if !done {
        if verbose then
            writeln("Adding locale 1’s contribution");
            total += computeMyContribution();
    }
}
```

Which outer-scope variables are referenced?
- `locale 0`
- `locale 1`
- `verbose`
- `total`
- `done`
CAPTURING VARIABLES FOR ON-CLAUSES

```chpl
config const verbose = false;
var total = 0,
    done = false;
...

on Locales[1] {  
    if !done {  
        if verbose then  
            writeln("Adding locale 1’s contribution");  
            total += computeMyContribution();  
    }  
}
```

Which outer-scope variables are referenced?
- `locale 0`
- `locale 1`

Which are constant? Can capture these by value.
- `verbose`
- `total`
- `done`

Argument bundle:
- `locale 0`
CAPTURING VARIABLES FOR ON-CLAUSES

onClause.chpl

```chpl
config const verbose = false;
var total = 0,
    done = false;

...

on Locales[1] {
    if !done {
        if verbose then
            printf("Adding locale 1’s contribution");
        total += computeMyContribution();
    }
}
```

Which outer-scope variables are referenced?

Which are constant? Can capture these by value.

Which are MCM-safe to cache? Can capture these by value.

MCM = Memory Consistency Model
CAPTURING VARIABLES FOR ON-CLAUSES

```
config const verbose = false;
var total = 0,
    done = false;

on Locales[1] {
    if !done {
        if verbose then
            writef("Adding locale 1’s contribution");
            total += computeMyContribution();
    }
}
```

Which outer-scope variables are referenced?

- `verbose`
- `total`
- `done`

Which are constant? Can capture these by value.
- `verbose`
- `total`
- `done`

Which are MCM-safe to cache? Can capture these by value.
- `locale 0`

What remains? Have these refer to the original, requiring R[D]MA to access.
- `locale 1`

MCM = Memory Consistency Model | R[D]MA = Remote [Direct] Memory Access
CAPTURING VARIABLES FOR ON-CLAUSES

```chpl
config const verbose = false;
var total = 0,
    done = false;
...

on Locales[1] {
    if !done {
        if verbose then
            writeln("Adding locale 1’s contribution");
        total += computeMyContribution();
    }
}
```
OTHER TASK PARALLEL FEATURES

- **begin / cobegin statements**: other ways of creating tasks
- **atomic / synchronized variables**: for sharing data & coordinating between tasks
- **task intents / task-private variables**: ways of controlling how variables relate to tasks
DATA PARALLELISM AND DOMAIN MAPS

Chapel language concepts

Domain Maps
Data Parallelism
Task Parallelism
Base Language
Locality Control
Target System

Higher-level Chapel
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);
DATA-PARALLEL ARRAY FILL

```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);
```

declare a domain, a first-class index set

declare an array over that domain
DATA-PARALLEL ARRAY FILL

```chapel
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);
```

- **Declare a domain, a first-class index set**
- **Declare an array over that domain**
- **Iterate over the domain's indices in parallel, assigning to the corresponding array elements**
DATA-PARALLEL ARRAY FILL

```chapel
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);
```

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly
DATA-PARALLEL ARRAY FILL

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);

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly
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);
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.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);
**DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)**

```chapel
fillArray.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);
```

apply a domain map, specifying how to implement...
...the domain's indices,
...the array's elements,
...the loop's iterations,
...on the program's locales
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);
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

```chapel
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);
```

```
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
```

```
locale 0
locale 1
locale 2
locale 3
```

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-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.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);
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

```chapel
goto fillArray.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);
```
fillArray.chpl

```chapel
... var D = {1..n, 1..n} dmapped Cyclic(...)=(1,1));
var A: [D] real;

forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
...
```
**IMPLEMENTING DATA PARALLELISM IN CHAPEL**

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

```
const Dmap = new Cyclic(startIdx=(1,1));
var D = Dmap.newDomain(idxs={1..n, 1..n});
var A = D.newArray(eltType=real);
for (i,j) in D.defaultParIterator() do
    A[i,j] = i + (j - 0.5)/n;
```

**Domain maps are written in Chapel using “lower-level” features**
- objects, methods, tasks, on-clauses, ...

```
iter CyclicDom.defaultParIterator() {
    coforall loc in this.targetLocales do
        on loc do
            coforall tid in 1..here.numPUs() do
                for idx in myInds(loc, tid, ...) do
                    yield idx;
}
```
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
```

- “steve”
- “lee”
- “sung”
- “david”
- “jacob”
- “albert”
- “brad”
CHAPEL-ENABLED OPTIMIZATIONS
**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):
STREAM TRIAD: C + MPI + OPENMP

```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );
    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );
    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf(outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose(outFile);
        }
        return 1;
    }

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }
    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;
}
```
STREAM TRIAD: CHAPEL (GLOBAL, PROMOTED VERSION)

use BlockDist;

config const m = 1000,
    alpha = 3.0;

const Dom = {1..m} dmapped Block({1..m});

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

B = 2.0;
C = 1.0;

A = B + alpha * C;

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] = 1.0;
        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;
}
STREAM TRIAD: CHAPEL VS. C+MPI+OPENMP

STREAM Performance (GB/s)

Locales (x 36 cores / locale)

- MPI+OpenMP
- Chapel EP
- Chapel Global

Better
STREAM TRIAD: CHAPEL (GLOBAL, RANDOM ACCESS VERSION)

```chapel
use BlockDist;

config const m = 1000,
                 alpha = 3.0;

const Dom = {1..m} dmapped Block({1..m});

var A, B, C: [Dom] real;
B = 2.0;
C = 1.0;

// was: A = B + alpha * C;
forall i in Dom do A[i] = B[i] + alpha * C[i];
```

Lots of overhead, given that we know all accesses are local
STREAM TRIAD: CHAPEL (GLOBAL, LOCAL ACCESS VERSION)

```plaintext
use BlockDist;

config const m = 1000,
    alpha = 3.0;

const Dom = {1..m} dmapped Block({1..m});

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

B = 2.0;
C = 1.0;

// was: A = B + alpha * C;
forall i in Dom do
    A.localAccess[i] = B.localAccess[i] + alpha * C.localAccess[i];
```

As fast as the promoted version, but... annoying
This optimization has been planned since the dawn of Chapel in the 2000's
  • motivated by work on ZPL in the 1990's
  • Implemented by Engin Kayraklioglu in June 2020
    • also supports several less-obvious variations
PERFORMANCE DUE TO AUTO-LOCAL-ACCESS OPTIMIZATION

STREAM

Bandwidth (TB/s)

Number of Locales (x 36 cores/locale)

Automatic Local Access

1.22

Better
var A, B, C: [Dom] real;
A = B + alpha * C;

var A, B, C: [Dom] real;
forall (a,b,c) in zip(A, B, C) do
  a = b + alpha * c;

var A, B, C: [Dom] real;
forall i in Dom do
  A[i] = B[i] + alpha * C[i];
HPCC RANDOM ACCESS (RA)

**Data Structure:** distributed table

**Computation:** update random table locations in parallel
/* Perform updates to main table.  The scalar equivalent is:

\[
\text{Ran} = (\text{Ran} \ll 1) \ ^ \oplus \ ((\text{Ran} < 0) \ ? \ \text{POLY} : 0);
\]

while (i < SendCnt) {
    for (i=0; i<NUPDATE; i++) {
        if (status.MPI_TAG == UPDATE_TAG) {
            Table[Ran & (tparams.TableSize - 1)] ^= Ran;
        } else if (status.MPI_TAG == FINISHED_TAG) {
            HPCC_Table[Localoffset] ^= Ran;
            HPCC_InsertUpdate(Ran, WhichProc, Buckets);
        }
    }
}

MPI_Init();
for (i=0; i<MAXUPDATE; i++) {
    Ran = (Ran << 1) ^ ((int) Ran < 0) ? POLY : 0;
} /* we got a done message.  Thanks for playing... */

MPI_Finalize();

} /* end receive done messages */

MPI_Finalize();

} /* MPIKernel */
/* 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 & (TABLESIZE-1)] ^= Ran;
}
*/
HPCC RA: CHAPEL VS. C+MPI (SEPTEMBER 2018)

RA Performance (GUPS)

GUPS

Locales (x 36 cores / locale)

Chapel 1.18.0
Chapel 1.17.1
Reference (bucketing)

better
for all (_, r) in zip(Updates, RAStream()) do
T[r & indexMask].xor(r);

gets lowered roughly to...

coforall loc in Updates.targetLocales do
on loc do
  coforall tid in 1..here.numPUs() do
    for idx in myInds(loc, tid, ...) do
      T[idx & indexMask].xor(idx);

  for idx in myInds(loc, tid, ...) do
    T[idx & indexMask].unorderedXor(idx);
    unorderedFence();

But, for a parallel loop with no data dependencies, why perform these high-latency operations serially?

- Implemented by Michael Ferguson and Elliot Ronaghan, 2019
HPCC RA: CHAPEL VS. C+MPI

RA Performance (GUPS)

GUPS

Locales (x 36 cores / locale)

Chapel
MPI

better
for all (\_, r) in zip(Updates, RAStream()) do
T[r & indexMask].xor(r);
HPC PATTERNS & BENCHMARKS

LCALS

Local loop kernels

NAS FT

Global Transposes

Global Random Updates

HPCC RA

Embarrassing/Pleasing Parallelism

Bucket-Exchange Pattern

Stencil Boundary Exchanges

STREAM Triad

ISx

PRK Stencil
HPC PATTERNS & BENCHMARKS: CHAPEL VS. REFERENCE

LCALS

NLALS: Chapel vs. Reference

NAS FT

NAS FT: Chapel vs. UPC vs. MPI

HPCC RA

HPCC RA: Chapel vs. C+MPI

HPCC STREAM Triad: Chapel vs. Reference

ISx: Chapel vs. Reference

PRK Stencil: Chapel vs. Reference

STREAM Triad

ISx

PRK Stencil

More on Chapel performance online at: https://chapel-lang.org/performance.html
NOTABLE APPLICATIONS OF CHAPEL

ChplUltra: Simulating Ultralight Dark Matter
Yale University / University of Auckland

CHAMPS: 3D Computational Fluid Dynamics
Eric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal

ChOp: Chapel-based Optimization
Nouredine Melab, Tiago Carneiro, et al.
INRIA Lille, France

Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DOD

CHGL: Chapel Hypergraph Library
Cliff Joslyn, Jesun Firoz, Louis Jenkins, et al.
PNNL

CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise

For more information, see: https://chapel-lang.org/poweredby.html
SUMMARY & RESOURCES
SUMMARY

Chapel cleanly and orthogonally supports...
...expression of parallelism and locality
...specifying how to map computations to the system

Chapel’s compilation is neither magical nor heroic
• lower-level features have straightforward compilation path
• higher-level features built in terms of lower-level features
• language designed with parallel optimization in mind

Chapel is powerful:
• supports succinct, straightforward code
• can result in performance that competes with, or beats, C+MPI+OpenMP

Chapel is attractive to computational scientists and Python programmers

[Image source: Kathy Yelick’s, CHIUW 2018 keynote: Why Languages Matter More Than Ever, used with permission]
CHALLENGES AND NEXT STEPS

• Generate code for GPUs
  • How will the compiler need to evolve? Will the language need to?

• Rearchitect the compiler
  • Shed cruft from research prototype days to harden the compiler
  • Reduce compile times
    – potentially via separate compilation / incremental recompilation?
  • Support interpreted / interactive Chapel programming

• Continue to optimize performance

• Release Chapel 2.0
  • guarantee backwards-compatibility for core language and library

• Continue to grow the Chapel community
WE ARE HIRING

Full-time:
• Keep an eye on our jobs site
  • (I need to update it this week)

Summers:
• HPE internships
• Google Summer of Code
CHAPEL RESOURCES

Chapel homepage: https://chapel-lang.org
- (points to all other resources)

Social Media:
- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: http://www.youtube.com/c/ChapelParallelProgrammingLanguage

Community / Support:
- Gitter: https://gitter.im/chapel-lang/chapel
- Discourse: https://chapel.discourse.group/
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues
SUGGESTED READING

Chapel Overviews / History:

- Chapel Comes of Age: Making Scalable Programming Productive, Chamberlain et al., CUG 2018, May 2018
- Proceedings of the 7th Annual Chapel Implementers and Users Workshop (CHIUW 2020), May 2020
- Chapel Release Notes — current version 1.22, April 2020

Implementation of Chapel’s High-Level Features:

- User-Defined Distributions and Layouts in Chapel: Philosophy and Framework, Chamberlain et al., HotPar’10, June 2010
- Authoring User-Defined Domain Maps in Chapel, Chamberlain et al., CUG 2011, May 2011
- User-Defined Parallel Zippered iterators in Chapel, Chamberlain et al., PGAS 2011, October 2011
Well-designed languages can improve user productivity while also enabling new optimizations.

We believe Chapel to be such a language.
THANK YOU

https://chapel-lang.org
@ChapelLanguage