ASYNCHRONOUS TASK-BASED AGGREGATED COMMUNICATION IN CHAPEL

Elliot Ronaghan
AMTE 2022
August 23, 2022
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
**Data Structure:** distributed table

**Computation:** in parallel, update random table elements with random values

**Declarations:** distributed table and index space of updates in Chapel:

```chapel
var T: [newBlockDom(0..<tableSize)] atomic int;
const Updates = newBlockDom(0..<numUpdates);
```
/* Perform updates to main table. The scalar equivalent is: */
/* */
/* for (int = 0; HMPDATE < HMPMAX; int++) */
/* Ran = (Ran < 32 ? (unsigned) Ran + 0 : POLY); */
/* ) */
/* */

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
while (i < SendCnt) {
    /* receive messages */
    Table[Ran & (TABSIZE - 1)] ^= Ran;
    while (i < SendCnt) {
        /* receive messages */
        Table[Ran & (TABSIZE - 1)] ^= Ran;
        if (have_done) |
            if (status.MPI_TAG == UPDATE_TAG) |
                if (GlobalOffset < (tparams_Top + tparams.MinLocalTableSize)) |
                    WhichPe = (GlobalOffset - (tparams.MinLocalTableSize + 1)); |
                    else WhichPe = (GlobalOffset - (tparams_Remainder) / tparams.MinLocalTableSize);
                    if (WhichPe == tparams.MyProc) |
                        LocalOffset = (Ran & (tparams.TableSize - 1)) - tparams.GlobalStartMyProc;
                        if (Ran = -1) |
                            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_NULL;
                                        pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize, tparams.dtype64, outreq);
                                        if (have_done) |
                                            pendingUpdates = peUpdates;
                                            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);
                                                else |
                                                    if (status.MPI_TAG == UPDATE_TAG) |
                                                        if (GlobalOffset < (tparams_Top + tparams.MinLocalTableSize)) |
                                                            WhichPe = (GlobalOffset - (tparams.MinLocalTableSize + 1)); |
                                                            else WhichPe = (GlobalOffset - (tparams_Remainder) / tparams.MinLocalTableSize);
                                                            if (WhichPe == tparams.MyProc) |
                                                                LocalOffset = (Ran & (tparams.TableSize - 1)) - tparams.GlobalStartMyProc;
                                                                if (Ran = -1) |
                                                                    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_NULL;
                                                                                pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize, tparams.dtype64, outreq);
                                                                                if (have_done) |
                                                                                    pendingUpdates = peUpdates;
                                                                                    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_NULL;
                                                                                            pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize, tparams.dtype64, outreq);
                                                                                            if (have_done) |
                                                                                                pendingUpdates = peUpdates;
                                                                                                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);
/* 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;
    }

forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
HPCC RA: CHAPEL VS. C+MPI

RA Performance (GUPS)

Locales (x 36 cores / locale)

Chapel
MPI

GUPS
14
12
10
8
6
4
2
0

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

/* receive messages */
Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);

if (have_done = (MPI_Test(outreq, &status, tparams.finish_tag) == MPI_REQUEST_NULL)) {
    /* we got a done message. */
    HPCC_Table[localOffset] ^= Rand;
    /* send garbage */
    do {
        MPI_Isend((Updates, proc_count * tparams.TableSize, MPI_COMM_WORLD, &status, tparams.finish_tag, MPI_COMM_WORLD, &outreq);
} while (/* send remaining updates in buckets */
    MPI_Waitall(proc_count, outreq);
}
else {
    /* send remaining updates in buckets */
    do {
        MPI_Isend((Updates, proc_count * tparams.TableSize, MPI_COMM_WORLD, &status, tparams.finish_tag, MPI_COMM_WORLD, &outreq);
} while (/* send remaining updates in buckets */
    MPI_Waitall(proc_count, outreq);
}

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

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);

        /* we got a done message. */
        HPCC_Table[localOffset] ^= Rand;
    }
else {
    /* send garbage */
    do {
        MPI_Isend((Updates, proc_count * tparams.TableSize, MPI_COMM_WORLD, &status, tparams.finish_tag, MPI_COMM_WORLD, &outreq);
} while (/* send remaining updates in buckets */
    MPI_Waitall(proc_count, outreq);
}
else {
    /* send garbage */
    do {
        MPI_Isend((Updates, proc_count * tparams.TableSize, MPI_COMM_WORLD, &status, tparams.finish_tag, MPI_COMM_WORLD, &outreq);
} while (/* send remaining updates in buckets */
    MPI_Waitall(proc_count, outreq);
}
HPCC RA: NAÏVE CHAPEL VS. C+MPI (SEPTEMBER 2018)

RA Performance (GUPS)

Locales (x 36 cores / locale)

Chapel 1.18.0
Chapel 1.17.1
Reference (bucketing)

GUPS

0 0.5 1 1.5 2 2.5 3

0 16 32 64 128 256

better
UNORDERED OPERATION OPTIMIZATION

Perform updates to main table. The scalar equivalent is:

```chapel
def unorderedFence():
    unorderedFence();
```

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

So, our compiler rewrites the inner loop to perform the ops asynchronously.

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

```chapel
forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
```
HPCC RA: CHAPEL VS. C+MPI (TODAY)

RA Performance (GUPS)

Locales (x 36 cores / locale)

GUPS

Chapel
MPI

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

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

Now, think about what it would take for a compiler to optimize the C+MPI code...

...or for a user to target the Cray XC’s network atomics manually (and portably?)

HPCC RA: CHAPEL VS. C+MPI
OUTLINE

I. Chapel by Example: HPCC RA
II. Chapel Motivation & Context
III. Tasking and Locality Features
IV. Chapel Aggregators
V. Arkouda and Aggregation
VI. Wrap-up
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

Why Consider New Languages at all?

| 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 |

• 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 strives 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]
CHAPEL BENCHMARKS TEND TO BE CONCISE, CLEAR, AND COMPETITIVE

STREAM TRIAD: C + MPI + OPENMP

use BlockDist;
config const m = 1000, alpha = 3.0;
const Dom = {1..m} dmapped ...;
var A, B, C: [Dom] real;
B = 2.0;
C = 1.0;
A = B + alpha * C;

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

HPCC RA: MPI KERNEL

STREAM Performance (GB/s)

Locales (x 36 cores / locale)

RA Performance (GUPS)

Locales (x 36 cores / locale)
NOTABLE CURRENT APPLICATIONS OF CHAPEL

Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DOD
~22k lines of Chapel

CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal
~120k lines of Chapel

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

CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise

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

Your Project Here?
1. Users should be able to program at high levels of abstraction and get good performance

   `Dst = Src[Inds]; // whole-array index gather`

2. Yet, when more control / better performance is needed, they can drop to lower levels...

   ```plaintext
   forall (d, i) in zip(Dst, Inds) do // parallel loop-based index gather
       d = Src[i];
   ```

   ...and even lower levels, as necessary...

   ```plaintext
   coforall loc in Dst.targetLocales do // explicit SPMD-style index gather
       on loc do
           forall i in Dst.localSubdomain do
               Dst.localAccess[i] = Src[Inds.localAccess[i]];  
   ```

   ...where “calling out to C / CUDA / MPI / etc.” is effectively the lowest level

3. Chapel builds its higher-level abstractions in terms of the lower-level ones to guarantee compatibility
CHAPEL’S MULTiresOLUTION FEATURE STACK

Chapel language concepts

Domain Maps
Data Parallelism
Task Parallelism
Base Language
Locality Control
Target System
CHAPEL’S “LOWER-LEVEL” FEATURES

Chapel language concepts

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

“Lower-level” Chapel
TASKING AND LOCALITY FEATURES
• Chapel tasks are asynchronous in sense that they are:
  • launched dynamically
  • managed by the runtime
  • no pre-determined start or end conditions
• Asynchrony gives expressiveness benefits in that any parallel pattern can be expressed
• In practice, many patterns use groups of tasks doing similar things
  • such as the coforall examples in this talk
• Asynchrony improves performance by spreading network load and avoids synchronization bottlenecks
  • aggregators in this talk benefit highly from this
**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 as a single task 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);
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  printf("Hello from task \%d of %d on %s\n", tid, numTasks, here.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.

```bash
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
```
**TASK-PARALLEL “HELLO WORLD”**

```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);
```

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly
**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);
```
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);
    }
}
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

```chpl
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);
    }
}
```

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

```
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
...```
Differences between Chapel and traditional PGAS / SHMEM

1. Chapel supports a post-SPMD execution model
   - **traditional PGAS**: all PEs/ranks/threads start by executing ‘main’
   - **Chapel**: a single task executes ‘main’ on locale 0 and additional parallelism* is introduced from there

(* = local or distributed)
const verbose = false;
var total = 0,
    done = false;

... on Locales[1] {
    if !done {
        if verbose then
            writeln("Adding locale 1’s contribution");
        total += computeMyContribution();
    }
}
1. Chapel supports a post-SPMD execution model
   - **traditional PGAS:** all ranks/threads/PEs start by executing ‘main’
   - **Chapel:** a single task executes ‘main’ on locale 0 and additional parallelism* is introduced from there
   
   (* = local or distributed)

2. Chapel’s partitioned global address space is also post-SPMD
   - **traditional PGAS:** “I have a variable named ‘x’, so you must too, and therefore I can refer to yours”
   - **Chapel:** “I see variable ‘x’ in my lexical scope, so I can refer to it, whether it’s local or remote”

One outcome of these differences is that Chapel feels much more like traditional programming
BULK COMMUNICATION IN CHAPEL: A TOOL FOR MANUAL AGGREGATION

```chapel
var Buff: [0..<buffSize] real;

on Locales[1] { 
  var LocBuff = Buff;
  processData(LocBuff);
  Buff = LocBuff;
}
```
CHAPEL AGGREGATORS
• Bale is a collection of mini applications and aggregation libraries
  • Chapel has several ports of Bale applications, including index gather
  • We use Bale to evaluate the productivity of our aggregators and to compare performance to SHMEM

• From the description in https://github.com/jdevinney/bale, Bale is a:
  “Vehicle for discussion for parallel programming productivity. The bale effort attempts to:
  – demonstrate some challenges of implementing interesting (i.e. irregular) scalable distributed parallel applications.
  – demonstrate an approach (aggregation) to achieve high performance for the internode communication in such applications
  – explore concepts that make it easier to write, maintain, and get top performance from such applications
  We use bale to evolve our thinking on parallel programming in the effort to make parallel programming easier, more productive, and more fun. Yes, we think making it fun is a worthy goal!”
use BlockDist, Random;

const numTasks = numLocales * here.maxTaskPar;
config const N = 1000000, // number of updates per task
    M = 10000; // number of entries in the table per task

const D = newBlockDom(0..<M*numTasks);
var Src: [D] int = D;
const UpdatesDom = newBlockDom(0..<N*numTasks);
var Dst, Inds: [UpdatesDom] int;

fillRandom(Inds, min=0, max=Src.size);

// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
// Naive index gather: $\text{Dst} = \text{Src}[\text{Inds}]$

forall $(d, i)$ in zip(Dst, Inds) do
  $d = \text{Src}[i]$;

‘Src’ is a distributed array with $\text{numEntries}$ elements

‘Dst’ and ‘Inds’ are distributed arrays with $\text{numUpdates}$ elements
BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

// Naive index gather: \( Dst = Src[Inds] \);

forall (\( d, i \)) in zip(Dst, Inds) do
  \( d = Src[i] \);

gets lowered roughly to...

coforall loc in Dst.targetLocales do
  on loc do
    coforall tid in 0..<here.maxTaskPar do
      for idx in myInds(loc, tid, ...) do
        D[idx] = Src[Inds[idx]];

A concurrent loop over the compute nodes
A nested concurrent loop over each node's cores
A serial loop to compute each task's chunk of gathers
BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

// Naive index gather: Dst = Src[Inds];
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];

Gets lowered roughly to...

coforall loc in Dst.targetLocales do
    on loc do
        coforall tid in 0..<here.maxTaskPar do
            for idx in myInds(loc, tid, ...) do
                D[idx] = Src[Inds[idx]];

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

for idx in myInds(loc, tid, ...) do
    unorderedCopy(D[idx], Src[Inds[idx]]);
    unorderedCopyTaskFence();

So, our compiler rewrites the inner loop to perform the ops asynchronously.

- Implemented by Michael Ferguson, 2019
// Naive index gather: $\text{Dst} = \text{Src}[\text{Inds}]$;

forall ($d, i$) in zip($\text{Dst}$, $\text{Inds}$) do
    $d = \text{Src}[i]$;
BALE INDEX GATHER KERNEL IN CHAPEL: AGGREGATOR VERSION

// Naive index gather
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];

use CopyAggregation;

// Aggregated index gather
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);

To use it, we simply replace the assignment with ‘agg.copy’

Give each task a “source aggregator”, agg, which aggregates remote ‘gets’ locally, then performs them

As the aggregator’s buffers fill up, it communicates the operations to the remote locale, automatically and asynchronously
use CopyAggregation;

// Aggregated index gather
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
Q: Is there an opportunity for the compiler to introduce aggregators automatically?

A: In many cases, yes

- developed by Engin Kayraklioglu, 2021
- combines previous ‘unordered’ analysis with a new locality analysis of RHS/LHS expressions
- for details, see Engin’s LCPC 2021 paper: [https://lcpc2021.github.io/](https://lcpc2021.github.io/)
As a result, the naïve version can now compete with the user-written aggregators

```plaintext
// Naive index gather: Dst = Src[Inds];
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];
```
**BALE INDEX GATHER: CHAPEL VS. EXSTACK VS. CONVEYORS**

**Elegant SHMEM version**

```plaintext
for (i = 0; i < N; i++)
    shmem_get(&target[i], &table[index[i]], sizeof(long), index[i] % NPES);
```

**Exstack version**

```plaintext
while (exstack_proceed(ex, (i == l_num_req)) ) {
    i0 = i;
    while (i < l_num_req) {
        l_indx = pckindx[i] >> 16;
        pe = pckindx[i] & 0xffff;
        if (!exstack_push(ex, &l_indx, pe)) break;
        i++;
    }
    exstack_exchange(ex);
    while (exstack_pop(ex, &idx, &fromth)) {
        idx = ltable[idx];
        exstack_push(ex, &idx, fromth);
    }
    lgp_barrier();
    exstack_exchange(ex);
    for (; i-0; i++ j++) {
        fromth = pckindx[j] & 0xffff;
        exstack_pop_thread(ex, &idx, [int64_t]fromth);
        tgt[i] = idx;
    }
    lgp_barrier();
}
```

**Conveyors version**

```plaintext
i = 0;
while (more = convey_advance(requests, (i == 1_num_req)), more | convey_advance(replies, !more)) {
    for (; i < 1_num_req; i++) {
        pkg.idx = i;
        pkg.val = pckindx[i] >> 16;
        pe = pckindx[i] & 0xffff;
        if (!convey_push(requests, &pkg, pe)) break;
    }
    while (convey_pull(requests, ptr, &from) == convey_OK) {
        pkg.idx = ptr->idx;
        pkg.val = ltable[ptr->val];
        if (!convey_push(replies, &pkg, from)) {
            convey_unpull(requests);
            break;
        }
    }
    while (convey_pull(replies, ptr, NULL) == convey_OK) {
        tgt[ptr->idx] = ptr->val;
    }
}
```

**Elegant Chapel version** (compiler-optimized w/ `--auto-aggregation`)

```plaintext
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
```

**Manually Tuned Chapel version** (using aggregator abstraction)

```plaintext
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
    agg.copy(d, Src[i]);
```
BALE INDEX GATHER: CHAPEL VS. EXSTACK VS. CONVEYORS

Cray XC (Aries)

bale index gather

Aggregate Throughput (GB/s)

Number of Locales (x 36 cores / locale)
CHAPEL’S AGGREGATORS

• Chapel’s aggregators are implemented as Chapel source code
  • no language or compiler changes were required
  • initial implementation only relied on high-level features
    – current optimized version calls into lower-level put/get routines

• Relies upon:
  – standard language features:
    – OOP: records, initializers, de-initializers
    – arrays
    – access to C-level pointers and dereferences
  – Chapel features that you’ve seen:
    – global namespace
    – task-local variables

• ~100 lines of reasonably straightforward code to implement SrcAggregator
  – (~420 lines for entire ‘CopyAggregation’ module)
record SrcAggregator {
    type elemType;
    var dstAddrs, srcAddrs: [LocaleSpace] [0..<bufferSize] addr;
    var bufferIdxs: [LocaleSpace] int;
    ...
}

proc flushBuffer(loc: int, ref bufferIdx) {
    var srcVals: [0..<bufferIdx] elemType;

    on Locales[loc] {
        const locSrcAddrs = srcAddrs[loc][0..<bufferIdx];
        var locSrcVals: [0..<bufferIdx] elemType;
        ...
        // fill the locSrcVals array
        srcVals = locSrcVals;
    }
    ...
    // assign the srcVals to the dstAddrs
}
INITIAL SRC AGGREGATOR IMPLEMENTATION: EXCERPTS

```plaintext
record SrcAggregator {
    type elemType;
    var dstAddrs, srcAddrs: [LocaleSpace] [0..<bufferSize] addr;
    var bufferIdxs: [LocaleSpace] int;
    ...
    proc flushBuffer(loc: int, ref bufferIdx) {
        var srcVals: [0..<bufferIdx] elemType;
        ...
        on Locales[loc] {
            const locSrcAddrs = srcAddrs[loc][0..<bufferIdx];
            var locSrcVals: [0..<bufferIdx] elemType;
            ... // fill the locSrcVals array
            srcVals = locSrcVals;
        }
        ... // assign the srcVals to the dstAddrs
    }
}
```

- Bulk array copy to 'get' the src addresses
- Bulk array copy to 'put' src values back to original locale
More flexible than traditional aggregators:

- **traditional aggregators**: like barriers or collectives, tend to assume everyone is involved and quasi-lockstep
- **Chapel aggregators**: Chapel’s post-SPMD nature relaxes traditional BSP constraints
  - tasks communicate with remote locales asynchronously, once a given buffer fills up
  - any subset of tasks/locales can utilize aggregators that target any locales *without those locales being involved*

User-level tasks make the implementation efficient

- Chapel leverages Sandia’s Qthreads

Performance is competitive with conventional techniques
Q: Clean code, competitive performance and scalability, no modifications to the language or compiler... 
...so, what’s the catch?

A: Not a ‘catch’ per se, but currently, Chapel’s aggregators only support copy-style and atomic operations

- Ultimately, want/need to support general operations (“user-defined aggregators”)
  - In principle, not so different from the existing ones
  - **Limiting factor:** These would most naturally be expressed with first-class functions (FCFs)
    ...but Chapel's support for FCFs is currently a bit weak

- That said, many interesting computations can be written with copy-style aggregation...
  ...like Arkouda!
ARKOUDA AND AGGREGATION
Motivation: Say you’ve got...
...HPC-scale data science problems to solve
...a bunch of Python programmers
...access to HPC systems

How will you leverage your Python programmers to get your work done?
ARKOUDA’S HIGH-LEVEL APPROACH

Arkouda Client (written in Python)

User writes Python code in Jupyter, making familiar NumPy/Pandas calls

Arkouda Server (written in Chapel)
ARKOUDA SUMMARY

What is it?

- A Python library supporting a key subset of NumPy and Pandas for Data Science
  - Uses a Python-client/Chapel-server model to get scalability and performance
  - Computes massive-scale results (multi-TB-scale arrays) within the human thought loop (seconds to a few minutes)
- ~22k lines of Chapel, largely written in 2019, continually improved since then

Who wrote it?

- Mike Merrill, Bill Reus, et al., US DoD
- Open-source: https://github.com/Bears-R-Us/arkouda

Why Chapel?

- high-level language with performance and scalability
- close to Pythonic
  - enabled writing Arkouda rapidly
  - doesn’t repel Python users who look under the hood
- ports from laptop to supercomputer
ARKOUDA GATHER

- For Arkouda's gather kernel, Chapel performance on a recent HPE Apollo system is well ahead of XC
  - These timings were taken in April 2021
  - System-level bugs hurt reference SHMEM performance, so no direct comparisons here
Arkouda Argsort at Scale

- Ran on a large Apollo system, summer 2021
  - 73,728 cores of AMD Rome
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - ~100 lines of Chapel code

Close to world-record performance—quite likely a record for performance/SLOC
WRAP-UP
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 Discussion / Support:
• Discourse: https://chapel.discourse.group/
• Gitter: https://gitter.im/chapel-lang/chapel
• Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
• GitHub Issues: https://github.com/chapel-lang/chapel/issues
**SUGGESTED READING / VIEWING**

**Chapel Overviews / History** (in chronological order):
- Chapel Comes of Age: Making Scalable Programming Productive, Chamberlain et al., CUG 2018, May 2018
- Proceedings of the 8th Annual Chapel Implementers and Users Workshop (CHIUW 2021), June 2021
- Chapel Release Notes — current version 1.24, April 2021

**Arkouda:**
- Arkouda GitHub repo and pointers to other resources: [https://github.com/Bears-R-Us/arkouda](https://github.com/Bears-R-Us/arkouda)

**CHAMPS:**
  - two of his students also gave presentations at CHIUW 2021, also available from the URL above
- Another paper/presentation by his students at [https://chapel-lang.org/papers.html](https://chapel-lang.org/papers.html) (search “Laurendeau”)
Chapel is designed for productive parallel programming at scale

- recent users have reaped these benefits in large applications

Though PGAS in nature, Chapel avoids SPMD / BSP assumptions

- parallelism is expressed in the source code starting from a single task
- lexical scoping simplifies PGAS-based communication
- the net result is a far more approachable distributed parallel language

For gather/scatter/sort in Arkouda and Bale, copy aggregators are key

- Chapel’s are implemented concisely and elegantly within the language
- performance rivals that of Exstack / Conveyors

Chapel’s design and language-based nature provide optimization benefits

- e.g., automatic asynchronous operations and automatic aggregation (as in Arkouda / Bale)
THANK YOU

https://chapel-lang.org
@ChapelLanguage