RECENT PERFORMANCE ADVANCES IN CHAPEL AND ARKOUDA

Brad Chamberlain, Elliot Ronaghan, Engin Kayraklioglu
OpenSHMEM 2021 keynote
September 16, 2021
OR: Multiresolution Support for Aggregated Communication in Chapel

Brad Chamberlain, Elliot Ronaghan, Engin Kayraklioglu
OpenSHMEM 2021 keynote
September 16, 2021
OR: CHAPEL! AGGREGATION!! LET’S GO!!!

Brad Chamberlain, Elliot Ronaghan, Engin Kayraklioglu

OpenSHMEM 2021 keynote
September 16, 2021
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
**HPCC RANDOM ACCESS (RA)**

**Data Structure:** distributed table

![Diagram of 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)] int;
const Updates = newBlockDom(0..<numUpdates);
```
/* Perform updates to main table. The scalar equivalent is: */
/* */
for (i=0; i<NUMDATA; 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+];
                LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                    tparams.GlobalStartMyProc;
                HPCC_Table[LocalOffset] ^= inmsg;
            }
        } else if (status.MPI_TAG == FINISHED_TAG) {
            /* Finish everyone else up... */
            MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
            while (have_done & NumberReceiving > 0);
            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+];
                        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 );
                }
            } else if (status.MPI_TAG == UPDATE_TAG) {
                MPI_Test(&outreq, have_done, MPI_STATUS_IGNORE);
                if (have_done) {
                    outreq = MPI_REQUEST_NULL;
                    pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                        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) {
                /* end garbage - who cares, no one will look at it */
                tparams.finish_len = 0;
                tparams.finish_tag = 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+];
                    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 );
            }
        } /* end receive messages */
        do {
            HPCC_InsertUpdate(Ran, WhichPw, Buffets);
            pendingUpdates++;
        } while (pendingUpdates > 0);
    } /* receive messages */
    else {
        MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
            MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
    }

    MPI_Test(&outreq, have_done, MPI_STATUS_IGNORE);
    if (have_done) {
        outreq = MPI_REQUEST_NULL;
        pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
            tparams.dtype64, (int)pe, UPDATE_TAG, MPI_COMM_WORLD, &outreq);
        pendingUpdates -= peUpdates;
    }
/* end receive messages */
for (proc_count = 0; proc_count < tparams.NumProcs; ++proc_count) {
    if (proc_count == tparams.MyProc & tparams.finish_req) {
        /* end garbage - who cares, no one will look at it */
        tparams.finish_len = 0;
        tparams.finish_tag = 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+];
            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 );
    }
} /* end receive messages */
for (proc_count = 0; proc_count < tparams.NumProcs; ++proc_count) {
    if (proc_count == tparams.MyProc & tparams.finish_req) {
        /* end garbage - who cares, no one will look at it */
        tparams.finish_len = 0;
        tparams.finish_tag = 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+];
            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 );
    }
} /* end receive messages */
/* 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;
 */

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

RA Performance (GUPS)

- Chapel
- MPI

Locales (x 36 cores / locale)

GUPS

better
HPCC RA IN CHAPEL: NAÏVE IMPLEMENTATION

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

Chapel Kernel

**Gets lowered roughly to...**

- 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 updates

```chapel
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);
```

/* Perform updates to main table. The scalar equivalent is:
 * for (j=0;j<MAX_UPDATE_TAG;)
 *   for (i=0;i<SendCnt;)
 *     if (PendingUpdates[i] == == UPDATE_TAG) {
 *       bufferBase+j ^= Ran;
 *     }
 */

while (i < SendCnt) {
  if (PendingUpdates[i] == == UPDATE_TAG) {
    bufferBase+j ^= Ran;
  }
  i++;
}

A serial loop to compute each task's chunk of updates

```chapel
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);
```
HPCC RA: NAÏVE CHAPEL VS. C+MPI (SEPTEMBER 2018)

RA Performance (GUPS)

Locales (x 36 cores / locale)

GUPS

Chapel 1.18.0
Chapel 1.17.1
Reference (bucketing)
UNORDERED OPERATION OPTIMIZATION

Chapel Kernel

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);

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?

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

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

RA Performance (GUPS)

Locales (x 36 cores / locale)

Chapel
MPI

greater
```plaintext
// Perform updates to main table. The scalar equivalent is:

HPCC_InsertUpdate(Run, WhishPb, BucketId);

foreach (r) in zip(Updates, RASTream()) do
  if (pe == HPCC_Table[Localoffset] ||
      tparams.StartMyProc == proc_count) { HPCC_Table[Localoffset] ^= inmsg;
    proc_count++; } else if (status.MPI_TAG == FINISHED_TAG) {
      proc_count += proc_count;
      MPI_Abort(MPI_COMM_WORLD, 0);
    }
  } while (have_done && NumberReceiving < 0);

MPI_Test(&inreq, shve-done, MPI_STATUS_IGNORE);
if (have_done) {
  outreq = MPI_REQUEST_NULL;
  MPI_Waitall(HPCC_Table[Localoffset] = inmsg);
  MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)pe, 404);
}
```

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?)
I. Chapel by Example: HPCC RA
II. Chapel Motivation & Context
III. “Low-level” Chapel 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

• 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, RELATIVE TO OTHER LANGUAGES

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**

```chapel
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;
```

**HPCC RA: MPI KERNEL**

```chapel
forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
```
NOTABLE CURRENT APPLICATIONS OF CHAPEL

**Arkouda: NumPy at Massive Scale**  
Mike Merrill, Bill Reus, et al.  
US DOD  
~16k lines of Chapel

**CHAMPS: 3D Unstructured CFD**  
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.  
École Polytechnique Montréal  
~48k lines of Chapel

**CrayAI: Distributed Machine Learning**  
Hewlett Packard Enterprise

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

**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

\[
\text{Dst} = \text{Src}[\text{Inds}]; \quad \text{// whole-array index gather}
\]

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

\[
\text{forall } (d, i) \text{ in zip(Dst, Inds) do} \quad \text{// parallel loop-based index gather}
\]
\[
d = \text{Src}[i];
\]

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

\[
\text{coforall loc in Dst.targetLocales do} \quad \text{// explicit SPMD-style index gather}
\]
\[
\text{on loc do}
\]
\[
\text{forall i in Dst.localSubdomain do}
\]
\[
\text{Dst.localAccess}[i] = \text{Src}[\text{Inds.localAccess}[i]];
\]

...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

“Lower-level” Chapel
“LOW-LEVEL”

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

‘here’ 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.
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);
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("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
            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

```
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
            writef("Adding locale 1’s contribution");
            total += computeMyContribution();
    }
}
DIFFERENCES BETWEEN CHAPEL AND TRADITIONAL PGAS / SHMEM

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
   Another is that Chapel has no real need for a symmetric heap
BULK COMMUNICATION IN CHAPEL: A TOOL FOR MANUAL AGGREGATION

```chapel
class bulkComm.chpl

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

on Locales[1] {
    var LocBuff = Buff;
    processData(LocBuff);
    Buff = LocBuff;
}
```

allocate an array on locale 0
move computation to locale 1
bulk 'get' from remote array
bulk 'put' to remote array
CHAMPS SUMMARY

What is it?
• 3D unstructured CFD framework for airplane simulation
• ~48k lines of Chapel written from scratch in ~2 years

Who wrote it?
• Professor Eric Laurendeau’s team at Polytechnique Montreal

Why Chapel?
• performance and scalability competitive with MPI + C++
• students found it more productive to use
  – senior students have reduced time-to-science
  – junior students can now accomplish in 3 months what had taken ~2 years
• achieving competitive results w.r.t. established, world-class frameworks from Stanford, MIT, etc.
override proc exchange(zone, bZone) {
    recvBuffer_ = bZone.remoteICGlobalIndex_.donorBuffer_;

    const haloSubDomain = bZone.haloSubDomain_;

    ref facetsCon = zone.facetsConnectivity_,
    elemCon = zone.elementsConnectivity_;

    for (haloIndex, iFacetTimesStride) in zip(haloSubDomain, 0.. by stride_) {
        elemCon.elementsGlobalIndex_[haloIndex] = recvBuffer_[iFacetTimesStride];
    }
}
use BlockDist, Random, CopyAggregation;

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, Rindex: [UpdatesDom] int;

fillRandom(Rindex, 208);
Rindex = mod(Rindex, M*numTasks);

// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

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

‘Src’ is a distributed array with numEntries elements

‘Dst’ and ‘Inds’ are distributed arrays with numUpdates elements
// Naive index gather
forall (d, i) in zip(Dst, Inds) do
d = Src[i];
// 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'

As the aggregator's buffers fill up, it automatically performs the communications with the remote locale

use CopyAggregation;

'with (var ...)' creates a variable per task that's executing the 'forall' loop

Here, we're giving each task a “source aggregator”, agg, which aggregates remote 'gets' locally, then performs them

'use' the module providing the aggregators
use CopyAggregation;

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

BALE INDEX GATHER KERNEL IN CHAPEL: AGGREGATOR VERSION
IMPLEMENTING CHAPEL’S AGGREGATORS

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

• 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)

• Developed by Elliot Ronaghan, 2020–present
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
    }
}

Arrays for buffering per-locale src/dst addresses
allocate a landing spot for the remote src values
Bulk array copy to ‘get’ the src addresses
Bulk array copy to ‘put’ src values back to original locale
Store src vals to dest addresses

time to flush a buffer?
move to the remote node to buffer src vals and copy them back

Arrays for buffering per-locale src/dst addresses
allocate a landing spot for the remote src values
Bulk array copy to ‘get’ the src addresses
Bulk array copy to ‘put’ src values back to original locale
Store src vals to dest addresses

Arrays for buffering per-locale src/dst addresses
allocate a landing spot for the remote src values
Bulk array copy to ‘get’ the src addresses
Bulk array copy to ‘put’ src values back to original locale
Store src vals to dest addresses

Arrays for buffering per-locale src/dst addresses
allocate a landing spot for the remote src values
Bulk array copy to ‘get’ the src addresses
Bulk array copy to ‘put’ src values back to original locale
Store src vals to dest addresses

Arrays for buffering per-locale src/dst addresses
allocate a landing spot for the remote src values
Bulk array copy to ‘get’ the src addresses
Bulk array copy to ‘put’ src values back to original locale
Store src vals to dest addresses

Arrays for buffering per-locale src/dst addresses
allocate a landing spot for the remote src values
Bulk array copy to ‘get’ the src addresses
Bulk array copy to ‘put’ src values back to original locale
Store src vals to dest addresses
INITIAL SRC AGGREGATOR IMPLEMENTATION: BULK PUT/GET OPS

```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 locSrcAddr = 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
CHAPEL AGGREGATORS: ATTRACTIVE PROPERTIES

- 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
AGGREGATION PERFORMANCE: CHAPEL VS. EXSTACK VS. CONVEY

- For Bale index gather, Chapel competes with Exstack on up to 256 nodes / 9k cores of Cray XC
  - These results are from October 2020
  - We have since done ~600-node runs with similar results
CAN WE AUTOMATE AGGREGATION?

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 from HPCC RA with a new locality analysis of RHS/LHS expressions
- for details, see Engin’s upcoming talk and paper at LCPC 2021, October 13–15: https://lcpc2021.github.io/

// 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]);
AUTO-AGGREGATION: IMPACT

- As a result, the naïve version can now compete with the user-written aggregators

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

![Graph showing aggregate throughput vs. number of locales](image)
**SOUNDS GREAT, WHAT’S THE CATCH?**

**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 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 FOR ARKOUDA**

**Motivation:** Say you’ve got...

...a bunch of Python programmers

...HPC-scale data science problems to solve

...access to HPC systems

How will you leverage your Python programmers to get your work done?

https://www.cscs.ch/computers/piz-daint/
Arkouda Client (written in Python)

Arkouda Server (written in Chapel)

 Writes Python code in Jupyter
 Invoking NumPy/Pandas ops
ARKOUDA SUMMARY

What is it?
- “A workbench for data science at massive scales and interactive rates”
  - massive scales = multi-TB data sets
  - interactive rates = within the human thought loop (seconds to few minutes)
- ~16k lines of Chapel, largely written in 2019, continually improved since then

Who did it?
- Mike Merrill, Bill Reus, et al., US DOD

Why Chapel?
- great distributed array support
- ports from laptop to supercomputer
- high-level language with C-comparable performance
- close to Pythonic—doesn’t repel Python users who look under the hood
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 Gather Performance
chpl 1.24.1 / ak 04/06/21 -- 8 GiB arrays
ARKOUDA ARGSORT: HERO RUN

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

Aggregators have been key to getting results like these
ARKOUDA AND AUTO-AGGREGATION

- In Arkouda, we removed all manual aggregation from the source and applied auto-aggregation
  - 61 loops in total
    - 39 are optimized automatically
    - 22 are not
  - 3 cases that were not using aggregators were auto-optimized

- The patterns where the aggregation does not fire:
  - 9: aggregation is not based on ‘forall’ loops
  - 6: compiler cannot prove that unordered operation is safe
  - 3: locality is hard to detect
  - 2: aggregated copy is not in the last statement of the body
  - 1: one side of the assignment is defined within the loop body
  - 1: needs further investigation
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:
• Gitter: https://gitter.im/chapel-lang/chapel
• Discord: https://chapel.discourse.group/
• 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:

- Bill Reus’s CHIUW 2020 keynote talk: https://chapel-lang.org/CHIUW2020.html#keynote
- Arkouda GitHub repo and pointers to other resources: https://github.com/Bears-R-Us/arkouda

CHAMPS:

- Eric Laurendeau’s CHIUW 2021 keynote talk: https://chapel-lang.org/CHIUW2021.html#keynote
  - 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 (search “Laurendeau”)
CHAPEL IS HIRING

- Chapel team at HPE is currently 18.5 full-time employees
  - planning to add 1–2 more during 2021
  - see: chapel-lang.org/jobs.html

- During summers, we also host interns and mentor Google Summer of Code students
Chapel is designed for productive parallel programming at scale
• recent users have reaped these benefits in 16k–48k-line 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 (as in RA) and automatic aggregation (as in Arkouda / Bale)