Co-Design Via Proxy Applications: MiniMD in Chapel

Brad Chamberlain, Ben Harshbarger, Chapel Team, Cray Inc.
SIAM PP14, MS78: Co-Design w/ Proxy Apps and Prog. Abstractions
February 21st, 2014
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.
What is Chapel?

- An **emerging parallel programming language**
  - Design and development led by Cray Inc.
    - in collaboration with academia, labs, industry
  - Initiated under the DARPA HPCS program

- A **work-in-progress**

- Chapel’s **overall goal**: **Improve programmer productivity**
  - Improve the **programmability** of parallel computers
  - Match or beat the **performance** of current programming models
  - Support better **portability** than current programming models
  - Improve the **robustness** of parallel codes
Chapel's Implementation

● Being developed as open source at SourceForge

● Licensed as BSD software

● A Community Effort
  ● version 1.8 saw 19 developers from 8 organizations and 5 countries

● Target Architectures:
  ● multicore desktops and laptops
  ● commodity clusters and the cloud
  ● HPC systems from Cray and other vendors
  ● *in-progress*: exascale-era architectures
Chapel and Proxy Applications

Overall, we like proxy applications a lot

- A chance to compare Chapel productivity/performance to status quo
- Users are more invested in them than traditional benchmarks
  - less likely to say “well, that’s nice, but it says nothing about my work”
  - more likely to wrestle through various design decisions with us
- Form a good basis for discussion between teams with distinct skill sets
  - codesign!
- Larger and more substantive than benchmarks
  - yet, without being overwhelming
- Documentation & reference versions have generally been pretty good
Chapel’s First DOE Proxy Application: LULESH

**Goal:** Solve one octant of the spherical Sedov problem (blast wave) using Lagrangian hydrodynamics for a single material

pictures courtesy of Rob Neely, Bert Still, Jeff Keasler, LLNL
Chapel’s First DOE Proxy Application: LULESH

History:
- Idea came about as a result of a Salishan discussion
- Summer intern did a naïve initial port to Chapel in a few weeks
- Chapel team made additional improvements over time
  - Included a productive pair-programming session with LLNL expert
- Now included as an example code in Chapel releases

Remaining work:
- Additional performance tuning work remains
  - Several general Chapel issues
  - Some application-specific (e.g., optimize data distribution for locality)
- Also, need to catch up with LULESH 2.0
LULESH in Chapel

| COMPUTE | STORE | ANALYZE |

LULESH in Chapel

Copyright 2014 Cray Inc.
LULESH in Chapel

1288 lines of source code
plus 266 lines of comments
487 blank lines

(the corresponding C+MPI+OpenMP version is nearly 4x bigger)

This is trunk/test/release/examples/benchmarks/lulesh/*.chpl in the SourceForge repository, as of r22745 (2/16/14).
LULESH in Chapel

This is all of the representation dependent code. It specifies:

- data structure choices
  - structured vs. unstructured mesh
  - local vs. distributed data
  - sparse vs. dense materials arrays
- their corresponding iterators
Here is some sample representation-independent code:

`IntegrateStressForElems()`
Representation-Independent Physics

```chapel
proc IntegrateStressForElems(sigxx, sigyy, sigzz, determ) {
forall k in Elems {
    var b_x, b_y, b_z: 8*real;
    var x_local, y_local, z_local: 8*real;
    localizeNeighborNodes(k, x, x_local, y, y_local, z, z_local);

    var fx_local, fy_local, fz_local: 8*real;

    local {
        /* Volume calculation involves extra work for numerical consistency. */
        CalcElemShapeFunctionDerivatives(x_local, y_local, z_local,
                                        b_x, b_y, b_z, determ[k]);

        CalcElemNodeNormals(b_x, b_y, b_z, x_local, y_local, z_local);

        SumElemStressesToNodeForces(b_x, b_y, b_z, sigxx[k], sigyy[k], sigzz[k],
                                    fx_local, fy_local, fz_local);
    }

    for (noi, t) in elemToNodesTuple(k) {
        fx[noi].add(fx_local[t]);
        fy[noi].add(fy_local[t]);
        fz[noi].add(fz_local[t]);
    }
}
```

All of this is independent of:
- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation
Multiresolution Design

**Multiresolution Design**: Support multiple tiers of features
- higher levels for programmability, productivity
- lower levels for greater degrees of control

*Chapel language concepts*

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

- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily
Data Parallelism in LULESH (Structured)

```plaintext
const Elems = {0..#elemsPerEdge, 0..#elemsPerEdge},
               Nodes = {0..#nodesPerEdge, 0..#nodesPerEdge};

var determ: [Elems] real;

forall k in Elems { ...determ[k]... }
```

![Diagram of Elems and Nodes](image-url)
const Elems = {0..#numElems},
Nodes = {0..#numNodes};

var determ: [Elems] real;

forall k in Elems { ...determ[k]... }
Materials Representation

- Not all elements will contain all materials, and some will contain combinations
Materials Representation (Dense)

naïve approach: store all materials everywhere (reasonable for LULESH 1.0, but not in practice)

```cpp
const Mat1Elems = Elems,
Mat2Elems = Elems;
```
Materials Representation (Sparse)

improved approach: use sparse subdomains to only store materials where necessary

```plaintext
var Mat1Elems: sparse subdomain(Elems) = enumerateMat1Locs(),
   Mat2Elems: sparse subdomain(Elems) = enumerateMat2Locs();
```
Q: How are domains and arrays implemented?  
(distributed or local?  distributed how?  stored in memory how?)

\[ \text{const } \text{Elems} = \{0..\text{numElems}\}, \]
\[ \text{Nodes} = \{0..\text{numNodes}\}; \]
\[ \text{var } \text{determin}: [\text{Elems}] \text{ real}; \]

A: Via domain maps…
Domain Maps: Concept

Domain maps are “recipes” that instruct the compiler how to map the global view of a computation...

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

...to the target locales’ memory and processors:
LULESH Data Structures (local)

```chapel
const Elms = {0..#numElems},
             Nodes = {0..#numNodes};

var determ: [Elems] real;

forall k in Elms { ... }
```

No domain map specified ⇒ use default layout
- current locale owns all indices and values
- computation will execute using local processors only
LULESH Data Structures (distributed, block)

```chapel
const Elems = {0..#numElems} dmapped Block(...),
Nodes = {0..#numNodes} dmapped Block(...);

var determ: [Elems] real;

forall k in Elems { ... }
```
LULESH Data Structures (distributed, cyclic)

```chapel
const Elems = {0..#numElems} dmapped Cyclic(...),
Nodes = {0..#numNodes} dmapped Cyclic(...);

var determ: [Elems] real;

forall k in Elems { ... }
```

Elems

Nodes
Chapel’s Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   ● to support common array implementations effortlessly

2. Advanced users can write their own domain maps in Chapel
   ● to cope with shortcomings in our standard library

3. Chapel’s standard domain maps are written using the same end-user framework
   ● to avoid a performance cliff between “built-in” and user-defined cases
Support compile-time reconfiguration

```plaintext
const ElemSpace = if use3DRepresentation
   then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
   else {0..#numElems},
NodeSpace = if use3DRepresentation
   then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
   else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
   else ElemSpace,
Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
   else NodeSpace;

const MatElems: MatElemsType = if useSparseMaterials then enumerateMatElems()
   else Elems;

proc MatElemsType type {
   if useSparseMaterials then
      return sparse subdomain(Elems);
   else
      return Elems.type;
}
```
MiniMD Study
What is MiniMD?

“Mini Molecular Dynamics”
- A proxy application from Sandia’s Mantevo group
- Representative of key idioms from real applications
- ~5000 lines of C++/MPI
  - ~2000 lines in Chapel

Molecular Dynamics?
- Computing physical properties like energy, pressure, and temperature for a simulated space containing moving atoms

Interesting in that it’s the first stencil code we’ve had a chance to focus on in Chapel
Store atoms in spatial bins

- Given a bunch of atoms...

  ```chapel
  record atom {
    var vel, force, position : 3*real;
  }
  ```

- Sort atoms into bins based on spatial position

  ```chapel
  const binSpace = {1..12, 1..12};
  var perBinSpace = {1..8};
  var bins: [binSpace] [perBinSpace] atom;
  ```

- Use cutoff to restrict number of atoms to compute against
  - Reduces complexity from $O(n^2)$ to $\sim O(n)$
Compute forces between atoms

```plaintext
forall bin in bins {
    for atom in bin {
        for neighbor in atom.neighbors {
            if distance(atom, neighbor) < cutoff {
                updateForces(atom, neighbor);
            }
        }
    }
}
```
Now let’s go to distributed memory…
Distributing Bins in C++/MPI

```c
while(ipx <= nprocs) {
    if(nprocs % ipx == 0) {
        nremain = nprocs / ipx;
        ipy = 1;

        while(ipy <= nremain) {
            if(nremain % ipy == 0) {
                ipz = nremain / ipy;
                surf = area[0] / ipx / ipy +
                      area[1] / ipx / ipz +
                      area[2] / ipy / ipz;

                if(surf < bestsurf) {
                    bestsurf = surf;
                    procgrid[0] = ipx;
                    procgrid[1] = ipy;
                    procgrid[2] = ipz;
                }
            }
            ipy++;
        }
    }
    ipx++;
}
```

```c
int reorder = 0;
MPI_Cart_create(MPI_COMM_WORLD, 3, procgrid,
                 periods, reorder, &cartesian);
MPI_Cart_get(cartesian, 3, procgrid, periods,
             myloc);
MPI_Cart_shift(cartesian, 0, 1, &procneigh[0][0],
                &procneigh[0][1]);
MPI_Cart_shift(cartesian, 1, 1, &procneigh[1][0],
                &procneigh[1][1]);
MPI_Cart_shift(cartesian, 2, 1, &procneigh[2][0],
                &procneigh[2][1]);

for(int idim = 0; idim < 3; idim++)
    for(int i = 1; i <= need[idim]; i++, iswap += 2) {
        MPI_Cart_shift(cartesian, idim, i,
                        &sendproc_exc[iswap],
                        &sendproc_exc[iswap + 1]);
        MPI_Cart_shift(cartesian, idim, i,
                        &recvproc_exc[iswap + 1],
                        &recvproc_exc[iswap]);
    }
```

+ Hundreds of lines of additional MPI setup
record atom {
    var vel, force, position : 3*real;
}

const binSpace = {1..12, 1..12};
var perBinSpace = {1..8};
var bins : [binSpace] [perBinSpace] atom;
Distributing Bins in Chapel

record atom {
    var vel, force, position : 3*real;
}

const binSpace = {1..12, 1..12} dmapped Block(...);
var perBinSpace = {1..8};
var bins : [binSpace] [perBinSpace] atom;
Compute forces between atoms (dist. mem.)

```plaintext
forall bin in bins {
    for atom in bin {
        for neighbor in atom.neighbors {
            if distance(atom, neighbor) < cutoff {
                updateForces(atom, neighbor);
            }
        }
    }
}
```

Runtime distributes work across locales and handles communication of data.
There must be a catch…?

Yes, performance! (today, at least)

- Chapel communication currently tends to be fine-grain, demand-driven
- Stencils really want to move slabs of data between neighbors
  - This is why stencils and MPI have had a positive feedback cycle

- Chapel was designed for good support for stencils…
  - See, for example, Richard Barrett’s CUG 2007 talk

…and for good stencil performance

- Based on previous work in ZPL which outperformed F+MPI for stencils

- Yet, stencils have not been a focus of our efforts to date
  - Sadly, HPCS milestones and HPCC have not required them…
ZPL scales better than MPI since its communication is expressed in an implementation-neutral way; this permits the compiler to use SHMEM on this Cray T3E but MPI on a commodity cluster.

ZPL also performs better at smaller scales where communication is not the bottleneck ⇒ new languages need not imply performance sacrifices.

Similar observations—and more dramatic ones—have been made using more recent architectures, languages, and benchmarks.
Generality Notes

Each ZPL binary supports:
- an arbitrary load-time problem size
- an arbitrary load-time # of processors
- 1D/2D/3D data decompositions

This MPI binary only supports:
- a static $2^{**k}$ problem size
- a static $2^{**j}$ # of processors
- a 3D data decomposition

The code could be rewritten to relax these assumptions, but at what cost?
- in performance?
- in development effort?
Code Size

Lines of Code

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>F+MPI</td>
<td>566</td>
</tr>
<tr>
<td></td>
<td>202 communication</td>
</tr>
<tr>
<td></td>
<td>242 computation</td>
</tr>
<tr>
<td>ZPL</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>70 declarations</td>
</tr>
<tr>
<td>A-ZPL</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>77 declarations</td>
</tr>
</tbody>
</table>

Copyright 2014 Cray Inc.
• the ZPL codes are 5.5–6.5x shorter because it supports a global view of parallelism rather than an SPMD programming model ⇒ little/no code for communication ⇒ little/no code for array bookkeeping

More important than the size difference is that it is easier to write, read, modify, and maintain
High-level languages can benefit Productivity

- more programmable, flexible
- able to achieve competitive performance
- more portable by leaving low-level details to the compiler

![Graph showing speedup over best 16-processor time (114.607 seconds in A-ZPL) for Cray T3E processors with different languages: F+MPI, ZPL, A-ZPL. The graph includes linear speedup and lines of code by language: communication (green), declarations (red), computation (blue).](image-url)
As ZPL, So Chapel?

ZPL-like results should be achievable by Chapel as well
  ● Chapel’s data parallel features are based on ZPL’s

Yet, Chapel lags ZPL precisely because of the generality introduced via abstractions like domain maps
  ● ZPL, like C and Fortran, “owned” its array format and operations
  ● Chapel permits it to be specified flexibly by the end-user
  ● Ultimately, similar performance should be achievable, but we started out with a significant disadvantage, and are still catching up

So what’s an impatient HPC programmer to do?
Use Chapel’s Multiresolution Features…

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
Use Chapel’s Multiresolution Features…

1) Ben wrote an explicit version of MiniMD
   ● SPMD + manually fragmented data structures as in an MPI code

   ```chapel
   class Chunk {...}
   var AllChunks: [LocaleSpace] Chunk;

   coforall loc in Locales do
       on loc {
           var myChunk = new Chunk(...);
           AllChunks[here.id] = myChunk;
           updateFluff(myChunk);
           forall bin in myChunk ...
       }
   ```

   ● of course, because of Chapel’s PGAS model, communication was expressed using array slicing rather than message passing
2) Then he refactored that logic into a Stencil domain map:
- an extension of Block supporting fluff and boundary conditions

```chapel
const binSpace = {1..12, 1..12} dmapped Stencil(...);
var perBinSpace = {1..8};
var bins : [binSpace] [perBinSpace] atom;
```

...with user-callable routines to update these values

```chapel
bins.updateFluff();
forall bin in bins {
    for atom in bin {
        for neighbor in atom.neighbors {
            if distance(atom, neighbor) < cutoff {
                updateForces(atom, neighbor);
            }
        }
    }
}
```
To browse MiniMD in Chapel

- See examples/benchmarks/miniMD/ in the Chapel release
- Or, point browser to:
  
  http://svn.code.sf.net/p/chapel/code/trunk/test/release/examples/benchmarks/miniMD

- You’ll find two versions of the code:
  - **version 1**: supports three approaches via compiler options:
    - single-locale (shared memory)
    - naïve multi-locale: uses Block distribution
    - Stencil-Block multi-locale: uses Ben’s custom distribution
  - **version 2**: explicit SPMD version
Next Steps

● Presently:
  ● working on single-locale optimizations to benefit most Chapel codes
Next Steps

● Presently:
  ● working on single-locale optimizations to benefit most Chapel codes

● Short-term:
  ● Detailed review of code for performance/elegance improvements
  ● Performance studies, comparisons, and optimizations
  ● Merge Stencil domain map capabilities into Block

● Longer-term:
  ● Have Chapel compiler automatically insert calls to update fluff
    ● (reproduce ZPL analysis and optimization within Chapel)
A Closing Note on Chapel’s Productivity

Ben...

- an undergraduate
- with no significant parallel programming experience
- no Chapel experience
- no MiniMD experience

...wrote 4 elegant versions of MiniMD in ~13 weeks

- 2 weeks: learned Chapel, miniMD, wrote single-locale transliteration
- 2 weeks: edited for Chapel style based on feedback from team
- 2 weeks: performance improvements and Block multi-locale version
- 3 weeks: explicitly distributed version
- 2.5 weeks: wrote the Stencil distribution version (and the dist. itself)
- 1.5 weeks: merged single-locale, Block, and Stencil versions into one
  - select between them with a compiler flag
Summary

● **Proxy apps are great**
  ● LULESH and MiniMD are particularly good examples

● **Initial Chapel ports of LULESH and MiniMD are available**
  ● Chapel’s programmability goals are being met
  ● more work required on performance optimizations and tuning
The Cray Chapel Team (Summer 2013)

Chapel USA

Chapel Seattle
Chapel...

...is a collaborative effort — join us!
For More Information: Online Resources

Chapel project page: http://chapel.cray.com
  ● overview, papers, presentations, language spec, …

Chapel SourceForge page: https://sourceforge.net/projects/chapel/
  ● release downloads, public mailing lists, code repository, …

Mailing Aliases:
  ● chapel_info@cray.com: contact the team at Cray
  ● chapel-announce@lists.sourceforge.net: announcement list
  ● chapel-users@lists.sourceforge.net: user-oriented discussion list
  ● chapel-developers@lists.sourceforge.net: developer discussion
  ● chapel-education@lists.sourceforge.net: educator discussion
  ● chapel-bugs@lists.sourceforge.net: public bug forum
For More Information: Suggested Reading

Overview Papers:

  - a high-level overview of the project summarizing the HPCS period
  - a more detailed overview of Chapel’s history, motivating themes, features

Blog Articles:

  - a series of technical opinion pieces designed to rebut standard arguments against the development of high-level parallel languages
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

Copyright 2014 Cray Inc.