Optimizing PGAS overhead in a multi-locale Chapel implementation of CoMD

Riyaz Haque and David F. Richards

LLNL-PRES-708978

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Acknowledgements

Our sincere thanks to

Tom MacDonald
Ben Albrecht
Ben Harshbarger
Brad Chamberlain

for their invaluable advice and timely help in optimizing various aspects of CoMD-Chapel

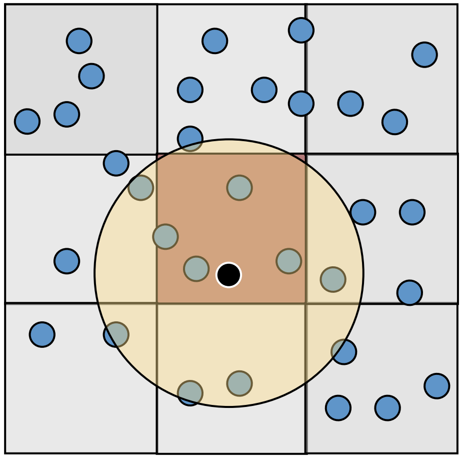
The CoMD proxy app

- Domain decomposition
  - Spatial *"linked-cell"* decomposition

- Force calculation
  - Iterate atom pairs and calculate contribution to energy & forces
  - *Lennard-Jones* and *EAM* potential energy models

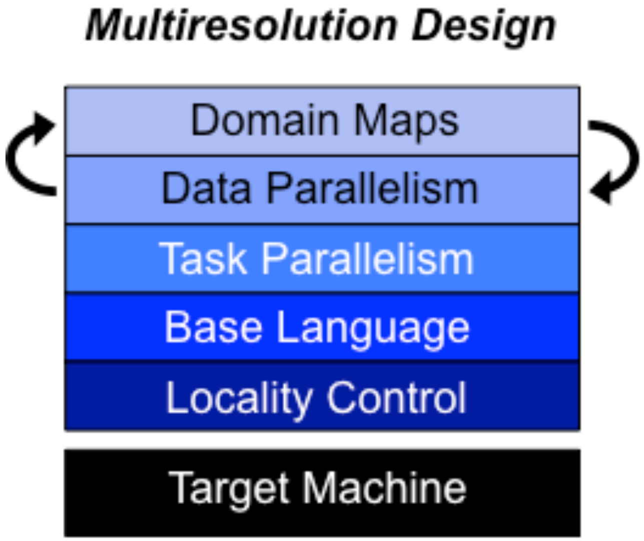
- Halo exchange communication
  - Data from adjacent nodes needed to compute forces on local particles
  - Data needed depends on potential used

- Implemented in several programming models
  - Our reference version based on MPI+OpenMP



The Chapel programming language

- A PGAS language
  - Implicit access of non-local data
- Abstractions for parallel and distributed computing
  - `coforall` and `forall` for concurrency
  - *Locales* as independent units of execution
  - Separation of concurrency and locality
    - `on` statement for switching execution between locales
- Extensive support for distributed arrays
  - Domains as index sets for arrays
  - Distributions for dividing domains and arrays across locales
    - Standard (e.g. `BlockDist`) and user-defined
  - Bulk array assignment for aggregating inter-locale communication



Multiresolution Design

Domain Maps

Data Parallelism

Task Parallelism

Base Language

Locality Control

Target Machine

Chapel is designed for productivity at scale

CoMD-Chapel implementation

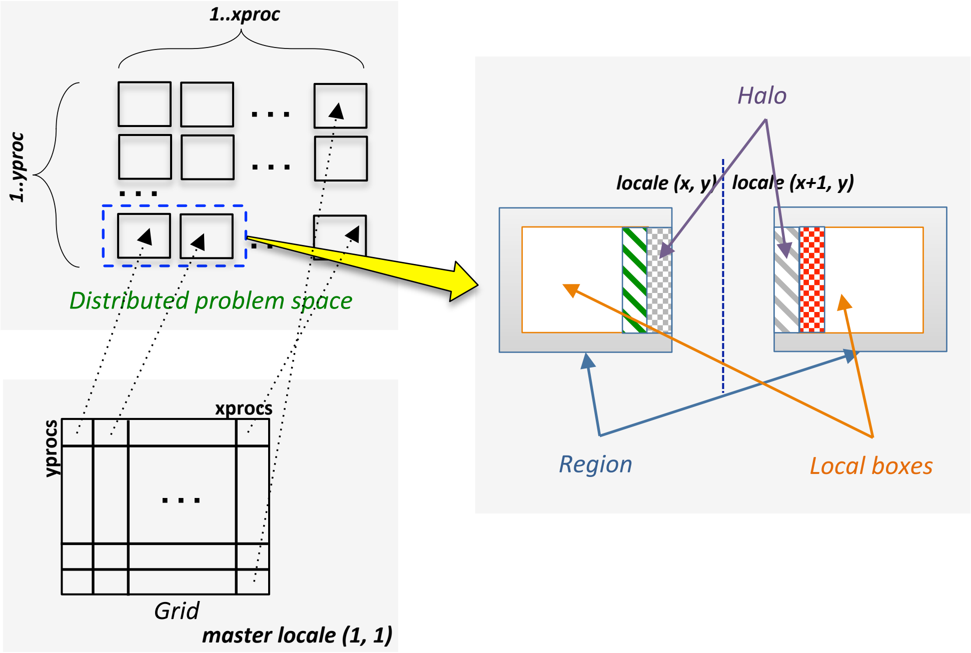
- Manually decompose the problem domain across all the locales
  - Simplifies analysis of data access patterns
  - To be abstracted away from application logic as a Chapel distribution

- Launching computation steps
  - SPMD manner of execution
  - Master locale creates independent tasks on other locales for each step
    - Simple, though not very scalable

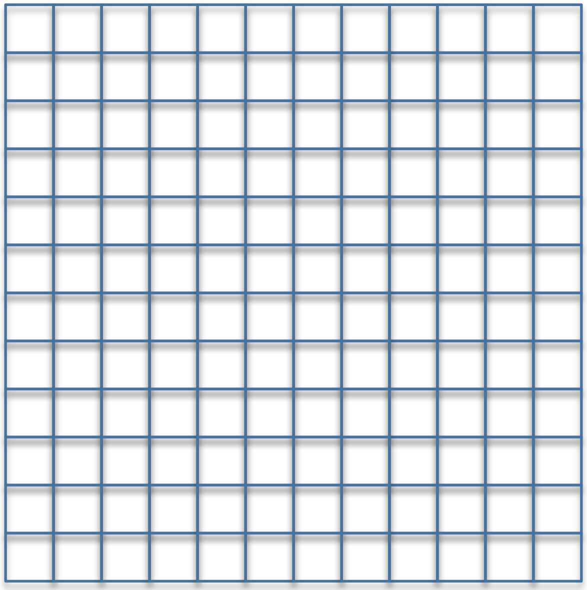
- A valid simulation preserves the total (potential + kinetic) energy of the system

The problem space is decomposed manually

- *Linked-cell* approach, similar to CoMD
  - Create an n-dimensional domain of locales (`locDom : domain(2) = {1..xproc, 1..yproc}`)
  - On each locale,
    - Decompose problem space into a set of local boxes (`localDom`) using Chapel’s `BlockDist`
    - Expand this set of local boxes in each dimension to accomodate the halo region (`localDom.expand(d)`)
  - On the master locale, create an array, `Grid`, corresponding to the expanded region on each locale

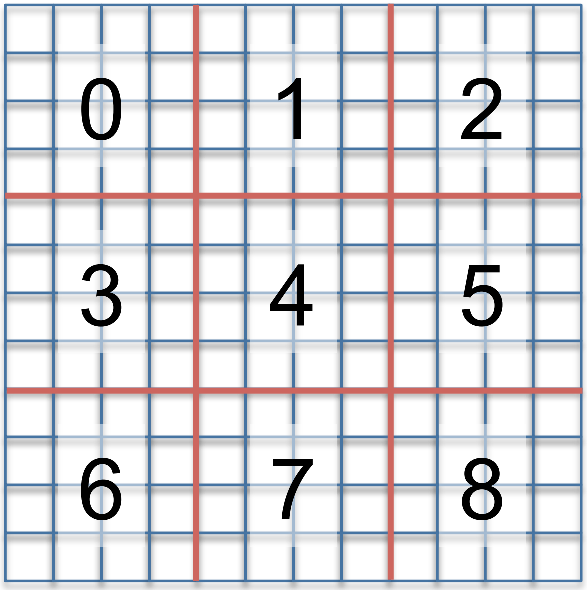


The problem space is decomposed manually



Start by breaking the problem domain into link cells

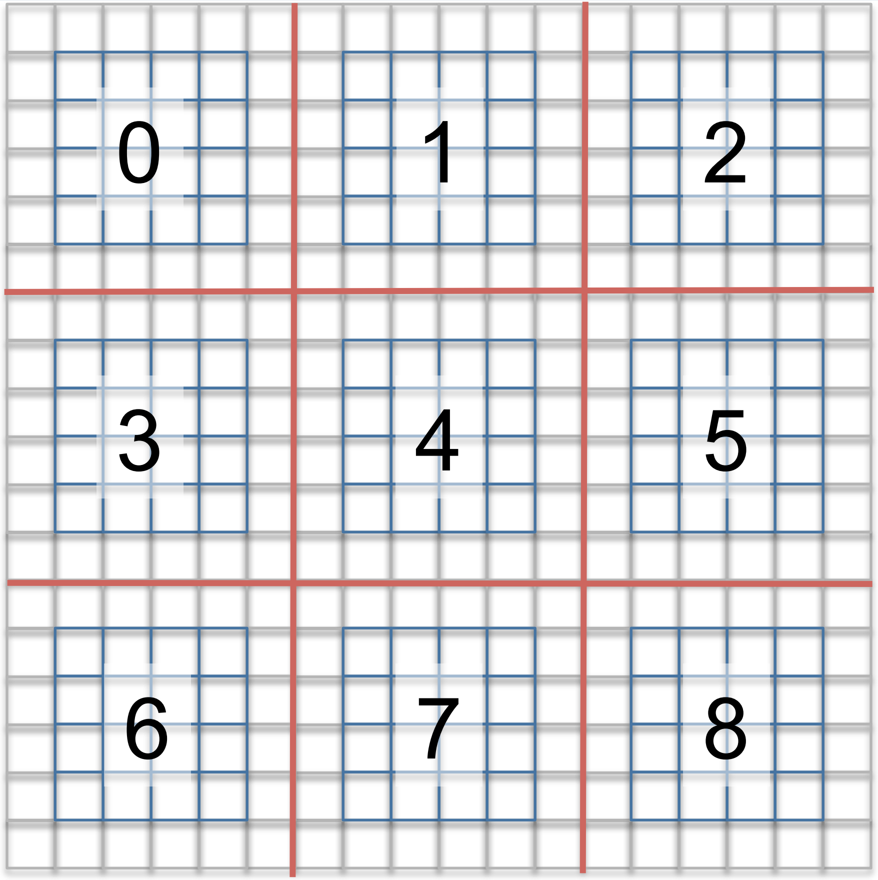
The problem space is decomposed manually



Start by breaking the problem domain into link cells,
and then divide the link cells between locales using Chapel BlockDist

This decomposition has terrible performance due to over-communication

The problem space is decomposed manually

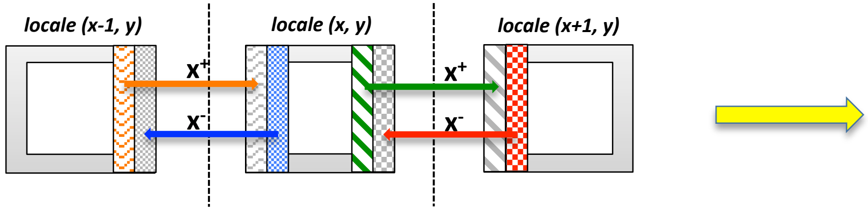


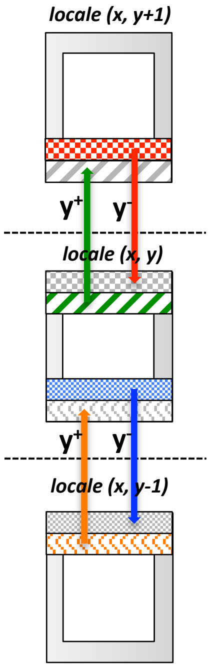
Add explicit storage for halos on each locale

This looks like an MPI decomposition. Glass half empty or half full?

Halo exchange minimizes inter-locale communication

- Exchange atoms along the faces
  - 2 data exchanges per dimension
  - Execute serially for each dimension





- Simultaneously update positions and apply periodic boundary conditions

- Aggregate data transfer using Chapel *bulk array assignment*

Inter-particle force computation is the dominant step

- CoMD-Chapel supports both *Lennard-Jones* and *EAM* potential energy models
  - force object stores force model parameters
  - fArr stores force data for particles

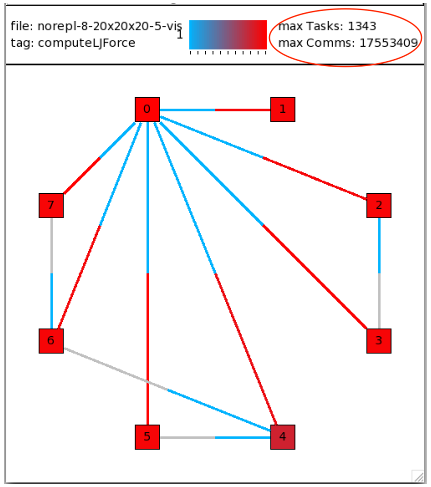
```
coforall box in cells[localDom] { // box
  for nBox in neighs[box] { // neighbor box
    for i in 1..box.count { // box atom
      for j in 1..nBox.count { // nBox atom
        if(dist(i, j) <= cutoff) {
          force.compute(i, j, fArr);
}}}}}
```

- *Full-neighbor* computation
  - Each atom computes all forces independently
  - *Half-neighbor* approach possible using atomics but not implemented

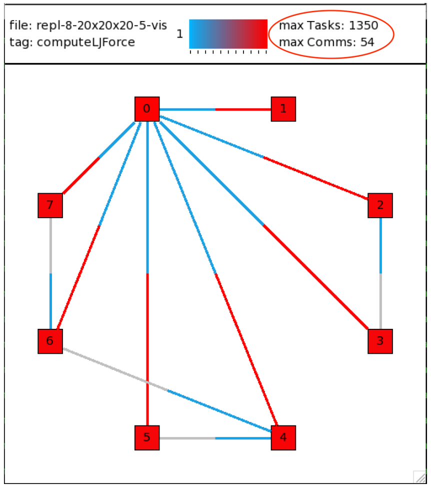
- Most compute-intense step of the application
  - Crucial to eliminate redundant inter-locale communication

Naïvely written code can result in unintended inter-locale communication

- The `force` object is initialized once on master locale; accessed each time-step on all locales
  - However, once created, it does not change
- Always referencing the copy on master locale results in extremely fine-grained communication
  - Severely hurts performance!
- Solution: Replicate the `force` object on each locale and access the local copy
  - ~1200x faster than the non-replicated version


file: norepl-8-20x20x20-5-vis
tag: computeLJForce

Without replication


file: repl-8-20x20x20-5-vis
tag: computeLJForce

With replication

Replication is an important optimization for CoMD-Chapel

Wide-pointers may be generated for data accessed locally

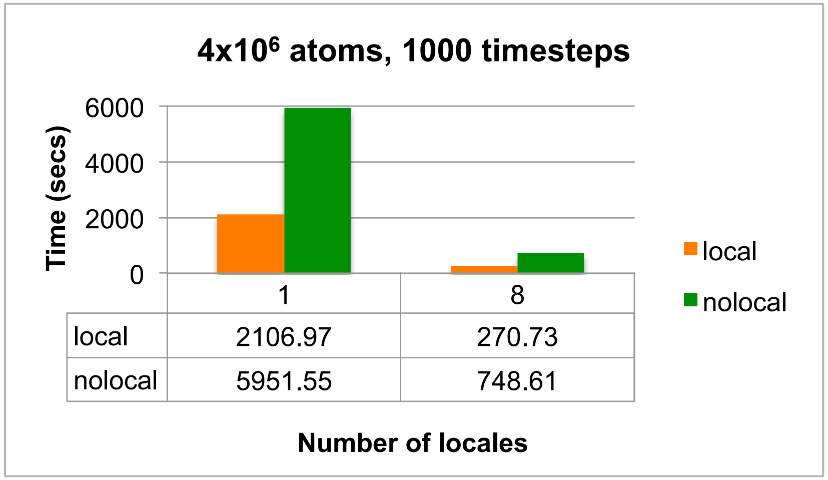
- Compiler may generate wide pointers for local data
  - Accessing local data through wide pointers incurs additional overhead

- Solution: Use `local` statement

```
local {
  // Access only local data
}
```

  - Compiler eliminates wide pointers for all references within the `local` block

Lower is better!

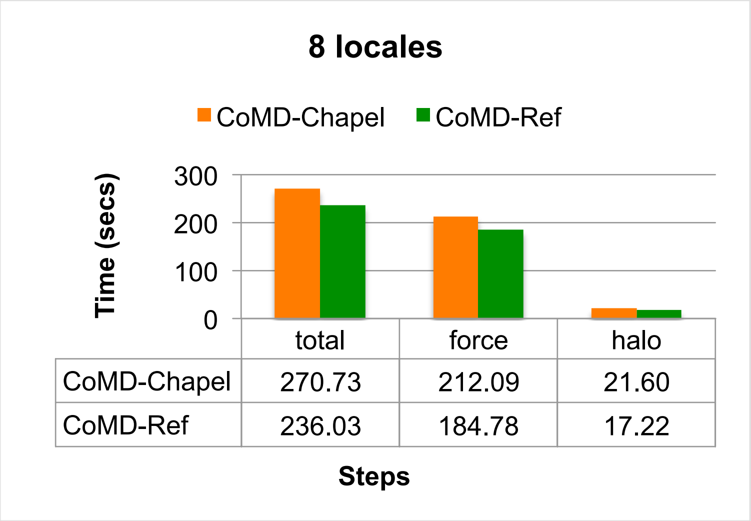


	1	8
local	2106.97	270.73
nolocal	5951.55	748.61

Localization results in a ~3x speedup for CoMD-Chapel

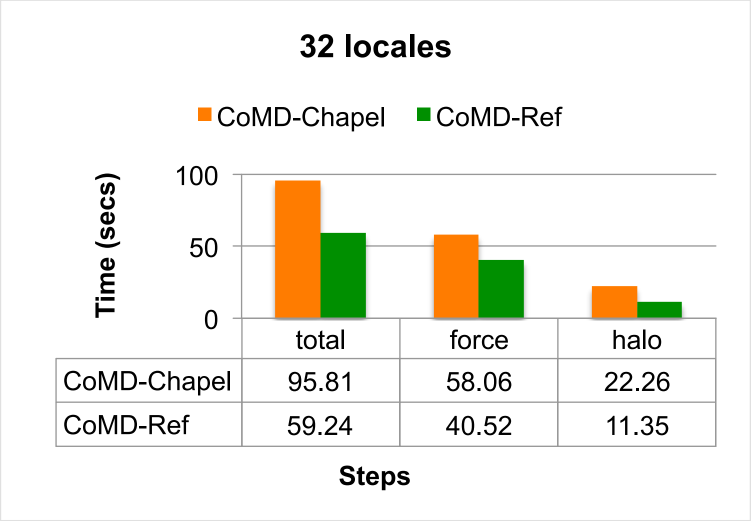
Performance is comparable to the reference implementation

- Code compiled using Chapel v1.13
  - The compiler itself was compiled with gcc-4.9.2, ibv on gasnet and qthreads as the threading framework
- Executed on 1-32 nodes of 64-bit Intel Xeon processors
  - 12 cores, 24 GB RAM, Infiniband high-speed interconnect



	total	force	halo
CoMD-Chapel	270.73	212.09	21.60
CoMD-Ref	236.03	184.78	17.22

Lower is better!



	total	force	halo
CoMD-Chapel	95.81	58.06	22.26
CoMD-Ref	59.24	40.52	11.35

CoMD-Chapel vs. CoMD-Ref, 4x10^6 atoms

CoMD-Chapel performs to within 87% (8 locales) to 67% (32 locales) of the reference

The local statement works for CoMD-Chapel, but …

- `local` statement is
  - Scope-based
    - May necessitate non-trivial code refactoring
  - Reactive
    - Programmer has to ensure *all* accesses are indeed local; possible runtime error otherwise
    - No compile-time checks
  - Restrictive
    - Does not work even if a single non-local access occurs in an arbitrarily large code block
    - May reject completely safe programs
  - Practical only for codes with visible and predictable access patterns

```
local { const x = f.m(); } // f.m() is local
func(x); // compile error! x is out of scope
```

```
local { const x = f.m(); }
// may fail at runtime if f.m() is modified
// to perform a non-local access
```

```
// all accesses are safe but may still throw
// a runtime error due to the on statement
local { /* on Locale(0) */
  x.m(); // x.m() is local to Locale(0)
  on(Locale(1)) { // switch to Locale(1)
    y.n(); // y.n() is local to Locale(1)
  }
}
```

The local statement has restricted applicability to complex real-life codes

Type-based (data-centric) locality is a more robust option

- Treats locality as an attribute of data rather than code blocks
  - Avoids scoping problems

```
const local x = f.m();
func(x); // no error since x is in scope
```

- Allows for more precise compile-time locality analysis

```
// compile-time error if f.m() returns a
// non-local reference, otherwise ok
const local x = f.m();
```

- Can support advanced locale topologies e.g. hierarchical places

```
// Possible hierarchical locale topology
local < level1 < level2 ... < global
```

- Can be expressed quite naturally within Chapel's rich type system

- Proposed for Chapel by Harshbarger [CHIUW 2015]
  - http://chapel.cray.com/CHIUW/2015/talks/data-centric-locality-chiuw-2015.pdf
  - More robust than asserting locality using the `local` statement
  - Implemented on a dev branch in Chapel v1.11

Data-centric locality should be made more widely available in Chapel

Summary

- Preventing unintended inter-locale communication is crucial for performance of CoMD-Chapel
  - Explicitly manage data decomposition and halo exchange
  - Force object requires replication on all locales
- Localization of wide pointers for local data is also very important
  - The `local` statement works for CoMD-Chapel but has serious limitations
  - Data-centric locality is a more robust approach
- With above optimizations in place, CoMD-Chapel performs between 69%-87% of the reference implementation
- Future work
  - Separating application logic from data distribution
  - Investigating strategies for improving scalability and reducing the task creation overhead for launching the computation steps
  - Replacing the `local` statement with data-centric locality analysis

Lawrence Livermore
National Laboratory

Questions