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

Riyaz Haque
Lawrence Livermore National Laboratory
Email: haque1@llnl.gov

David Richards
Lawrence Livermore National Laboratory
Email: richards12@llnl.gov

***Abstract*—Chapel supports distributed computing with an underlying PGAS memory address space. While it provides abstractions for writing simple and elegant distributed code, the type system currently lacks a notion of *locality* i.e. a description of an object's access behavior in relation to its actual location. This often necessitates programmer intervention to avoid redundant non-local data access. Moreover, due to insufficient locality information the compiler ends up using "wide" pointers—that can point to non-local data—for objects referenced in an otherwise completely local manner, adding to the runtime overhead.**

In this work we describe *CoMD-Chapel*, our distributed Chapel implementation of the CoMD benchmark. We demonstrate that optimizing data access through *replication* and *localization* is crucial for achieving performance comparable to the reference implementation. We discuss limitations of existing scope-based locality optimizations and argue instead for a more general (and robust) type-based approach. Lastly, we also evaluate code performance and scaling characteristics. The fully optimized version of CoMD-Chapel can perform to within 62%-87% of the reference implementation.

***Index Terms*—Chapel, CoMD, PGAS, Locality, Distributed computing, Parallel programming**

1. Introduction

Chapel [1], [2], [3] is a PGAS language with support for distributed programming. A Chapel *locale* is an independent unit of execution and associated memory analogous to an MPI rank. The `on` statement switches execution to a different locale. Due to the PGAS nature of the address space, it is possible to create objects on one locale and refer to them from another; inter-locale communication is, however, expensive. Thus reasoning about locality is critical when referencing data in a multi-locale setup. The following Chapel snippet illustrates the point

```
1 var f:F = new F(); // create f
2 on (newLocale()) { // switch to a new locale
3   ...
4   f.m(); // dereference f
5   ...
6 }
```

Code 1. PGAS behavior

This code creates an object `f` of type `F` on the current locale (line 1), switches to another (possibly the same) locale (line 2) and then dereferences `f` at the new locale (line 4). Depending on application logic, it may be permissible for `f` to be *replicated* (and henceforth accessed locally) across all locales eliminating remote communication albeit at a potential cost of extra coding on part of the programmer. Moreover, the compiler might conservatively use a wide pointer to `f` even if the current and target locales are the same, increasing the runtime overhead of accessing `f`. To mitigate this, `f.m()` can be enclosed in a `local` block which enables the Chapel compiler to generate only local pointers for the enclosed code. This *localization* asserts that any enclosed statements can only access local memory. Clearly, this approach fails if even a single expression inside `f.m()` has non-local semantics and its generally difficult to isolate exclusively local sections within a non-trivial code. Incorporating locality information within `f`'s type (line 1) can help alleviate these problems, enable a more precise compiler analysis and result in better optimized code.

CoMD [4] is one of several proxy apps for the classical molecular dynamics simulation problem. It has a simple decomposition of the problem space and a well-defined separation of computation and communication steps. This makes it easier to analyze object access patterns. Hence we have used CoMD for our work.

This paper makes the following contributions

- An optimized distributed Chapel implementation of the CoMD proxy app that performs to within 62%-87% of the reference implementation
- Demonstration of the importance of locality optimizations for achieving reasonable performance
- Performance evaluation of CoMD-Chapel under different optimizations and a comparison with the reference implementation
- Discussion on the limitations of the current language support for scope-based code locality and a language extension proposal for the more general type-based data locality approach

The rest of the paper is organized as follows. Section 2 briefly describes relevant Chapel terminology and the CoMD benchmark, Section 3 describes the CoMD-Chapel implementation in detail, Section 4 describes optimizations for reducing the PGAS overhead, Section 5 provides the

978-1-5090-5214-1/16 $31.00 © 2016 IEEE
DOI 10.1109/PAW.2016.9

rationale for type-based locality, Section 6 evaluates code performance, Section 7 explores related work and Section 8 discusses conclusions and future work.

2. Background

2.1. Chapel

We now briefly describe relevant Chapel constructs [3].

A Chapel *locale* is an independent unit of execution and associated memory. Execution starts on a master locale. The `on` statement creates a remote task on the target locale.

Chapel also provides constructs for parallelism. A `coforall` loop creates a thread for every instance of the loop. A `forall` loop creates a thread for every core on the locale and loop instances are share these threads. `begin` and `cobegin` blocks create asynchronous tasks for the enclosed statements. Note that parallelism and distributed execution are orthogonal concepts in Chapel and are often used together to obtain asynchronous distributed code (Code 3)

A Chapel `domain` is a multi-dimensional set of indexes. An *array* is created over a domain

```
var dom : domain(3) = {0..1, 0..1, 0..1};
var arr : [dom] int;
```

Domains can be distributed across locales using a *distribution*. An array defined over a distributed domain is also distributed across the locales. Some standard distributions are available in the Chapel runtime e.g. `BlockDist`;

```
var dDom : domain(3) = dom dmapped Block();
var dArr : [dDom] int; // distributed array
```

Chapel supports bulk array assignment that copies an array using aggregate communication thereby reducing overhead [5]. We use this feature to implement data exchange in CoMD-Chapel.

Chapel also supports object-oriented programming. `class` and `record` types define an object with associated state and member functions. The former has reference semantics and supports inheritance while the latter has value semantics and does not support inheritance.

Synchronization is done using *sync* variables and the `reduce` clause implements reduction. The language also has support for generic functions constrained by the `where` clause; such a function is instantiated conditionally based on the evaluation of the where clause.

2.2. CoMD

CoMD is a classical molecular dynamics proxy app that supports two potential energy models, *Lennard Jones (LJ)* and *Embedded Atom Method (EAM)*. During each timestep, inter-atom forces governed by the potential energy function are calculated and atoms are moved to new positions accordingly. Each atom can interact only with other atoms located within a cutoff region defined by the potential function. This property enables a very natural *linked-cell* partitioning of the problem space into *domains*; each domain owns the atoms within its "*local*" part of the overall simulation space. A domain also possesses a *halo*—a slice of each neighboring domain containing atoms that can interact with its own atoms. Each domain proceeds independently except at one point during the timestep when neighboring domains need to update halo atom positions and potentially exchange atoms that have moved into each others space. In CoMD, atom position update and exchange is implemented as a single communication step. The nearest image calculations required for periodic boundary conditions are handled in the halo exchange step as well.

CoMD has been implemented in many different programming models for instance MPI+X, CUDA, OpenCL, OpenACC, X10 etc. For our work we use the MPI+OpenMP based CoMDv1.1 [6] as the reference implementation.

3. Implementation

3.1. Domain decomposition

Like CoMD, CoMD-Chapel is also based on the *linked-cell* decomposition where the problem space is partitioned into `Boxes` of `Atoms`.

```
record Atom { // atom info
  var id : int(32); // id
  var r  : real3;   // position
  var v  : real3;   // velocity
}

record Box { // box of atoms
  var count : int(32); // number of atoms
  var atoms : [1..MAXATOMS] Atom; // atoms
}
```

Code 2. Atoms and Boxes

The problem space shown in Figure 1 is organized as an array (`Grid`) of wide pointers to objects (`Region`) located on each locale containing a set of `Boxes` corresponding to that locale's local and halo cells. We create these data structures manually at program initialization.

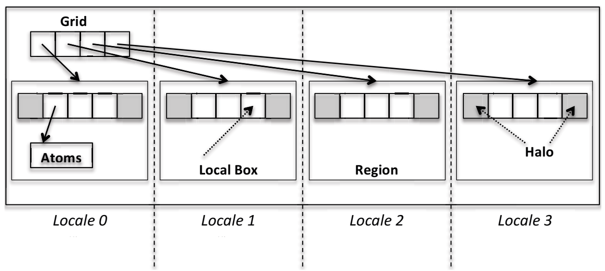


Figure 1. CoMD-Chapel - Domain decomposition

Chapel has a rich set of operations on arrays and domains. This greatly facilitates the task of grid initialization e.g. given the domain of local boxes on a locale, `localDom`, the corresponding halo region of a desired width `w` boxes in each dimension can be obtained as

```
halo : domain(3) = localDom.expand(w)
```

The choice of manually applying an MPI-like problem decomposition across locales instead of automatic partitioning using standard Chapel distributions is a conscious one. The standard Chapel `BlockDist` cannot automatically handle halo regions. The Chapel stencil distribution (`StencilDist`) works for halos but it cannot be integrated in our code without substantial a code rewrite. Most importantly however, using a standard distribution would obscure data access patterns within the code; something we wish to avoid at present in order to keep program analysis straightforward. We intend to eventually implement the code using the stencil distribution but that is beyond the scope of this work.

The grid is initialized in a way that each locale gets assigned an equivalent amount of work.

3.2. Computation steps

CoMD-Chapel effectively executes in a SPMD manner. Computation starts on the *master* locale. Each timestep executes a sequence of steps where the master locale forks independent tasks on all the locales by iterating over the `Grid` as follows (except halo exchange, see Section 3.2.4)

```
... // on master locale
coforall region in Grid { // for each region
  on (region.locale) { // dest locale
    ... // execute task
  }
}
```

Code 3. Step tasks

This approach is clearly not scalable. As we see later in Section 6, it introduces noticeable runtime overhead. However, it is the easiest to implement and for this paper we are primarily concerned with studying data access optimizations; we leave the design of a scalable task creation strategy to future work.

Each task internally uses `coforall` and `forall` for further exploiting intra-locale parallelism; analogous to creating threads on an MPI rank. Following each step, execution converges back at the master locale which then forks tasks for the next step. The length of the timestep interval remains constant for the duration of the program. A correct simulation conserves the total (potential + kinetic) energy of the system.

The simulation involves the following steps

3.2.1. Force computation.

Force computation is the most compute-intense part of the entire code. CoMD-Chapel also supports the LJ and the EAM potential functions. EAM is more complex of the two and requires additional communication between neighbors. The force model can quite naturally be represented as a class. This also ensures future extensibility using inheritance. A `force` object is created at program initialization on the master locale to store information about the desired model. For purpose of this paper we restrict ourselves to the simpler LJ model.

Since force equations are symmetric, it is sufficient to calculate the force once per pair of atoms and update each atom's total forces. This *half-neighbor* approach requires locking to prevent concurrent updates. We instead follow the simpler, but computationally heavier, *full-neighbor* approach where each atom calculates all its forces independently. The force array `fArr` stores the forces on each atom.

During each timestep, force computation proceeds on every locale as follows

```
1 coforall box in cells[localDom] { // box
2   for nBox in neighs[box] { // neighbor box
3     for i in 1..box.count { // box atom
4       for j in 1..nBox.count { // nBox atom
5         if(dist(i, j) <= cutoff) {
6           force.compute(i, j, fArr);
7 }}}}}
```

Code 4. Force loop

Note that application logic by itself does not require any communication inside the force loop (Code 4, lines 5-7.) However, in a naive implementation all locales repeatedly call (line 6) the original `force` object on the master locale hurting performance. To fix this, we replicate the `force` object across all the locales. Section 4.1 further elaborates on this critical optimization.

3.2.2. Position/velocity update.

These steps compute the new velocities and positions for the atoms in a leapfrog manner. The velocity is calculated twice per timestep in increments of one-half the time interval. This is the least compute-intensive part of the code. Just like force, this step also requires no inter-locale communication.

3.2.3. Energy calculation.

For ensuring simulation validity, the total energy of the system is checked for conservation every few timesteps. This requires a reduction across all atoms which is accomplished using Chapel's `reduce` clause.

3.2.4. Halo exchange.

Like CoMD, a region in CoMD-Chapel also updates the atom positions and exchanges the atoms in the same step. This is done for each locale by communicating with its six neighbors in three serial tasks, first along the two x-faces, then the two y-faces and finally both the z-faces. The halo exchange tasks are created as follows

```
... // on master locale
for i in x,y,z { // for each axis
  coforall region in Grid { // for each region
    on (region.locale) { // dest locale
      // exchange atoms along faces
      exchange(i.m, i.p);
}}}
```

Code 5. Halo exchange tasks

The `exchange` step is implemented using a *push-notify* mechanism. During the exchange step, each face uses bulk array assignment to *push* data to a neighbor's array and then *notifies* that neighbor by setting a corresponding sync variable. Upon notification, the neighbor updates its state from the copied array. For two faces along an axis, data is copied concurrently using `cobegin`. Additionally, if the thickness (number of boxes in a dimension) of a region is larger than one, then the region's state can also be updated for both the faces in that dimension concurrently. We implement this optimization by templatizing the `exchange` function with a `where` clause.

4. Optimization

Despite being written in a style close to the reference implementation, CoMD-Chapel needs two important optimizations in order to achieve comparable performance. These are replication for minimizing redundant remote accesses and localization for preventing creation of wide pointers for local data. We now describe each one in detail.

4.1. Replication

Chapel's PGAS semantics imply that the runtime will automatically insert communication operations to retrieve any non-local data being accessed on the current locale. Thus a Chapel object created on one locale can be (implicitly) accessed through its reference from another. Refer to our force loop in Code 4. The `force` object, instantiated on the master locale at program start-up time, is repeatedly referenced inside the force computation loop as shown in line 6. We would like to remind (Section 3.2.1) that the `force` object stores parameters of the potential function and is different from the force array `fArr` which stores the actual forces on the particles.

Naively referencing the version of the `force` object on the master locale multiple times results in numerous extremely fine-grained messages. We illustrate this with the following example.

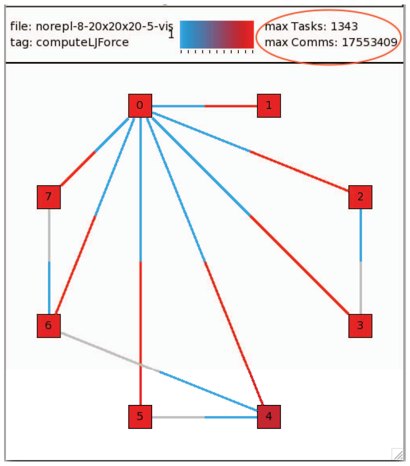


Figure 2. CoMD-Chapel - Without force replication

Consider a simulation of 32000 atoms distributed across 8 locales, running for 5 timesteps.

Figure 2 shows the inter-locale communication for the force computation step in an unoptimized implementation where all locales always access the force object on the master locale. The output is generated using the Chapel visualization and profiling tool, *chplvis* [7]. The circled number is a lower bound on the number of interactions between locales; such a high value for a step where no inter-locale communication is expected clearly shows the problem with disregarding object locality in Chapel.

Since the `force` object does not change post-initialization, it is easiest to simply replicate it on all locales and use only the local copy of the object. We have written a generic class `Replicated` parametrized on the type of the replicated object which needs to provide a callback method `replicate()`. A reference to the local copy of the object can be obtained using the `getLocalRef()` method.

```
class Replicated {
  type eltType; // must define replicate()
  proc Replicated(type eltType, tvar : eltType)
  proc getLocalRef() : eltType
}
```

Code 6. Replicated class

Internally `Replicated` uses the native Chapel distribution `ReplicatedDist` to create object copies on all locales. In this optimized version, the force object declaration becomes

```
var force = new Replicated(Force, new Force())
```

while the loop in Code 4 is transformed as

```
1 var fLocal = force.getLocalRef(); //get local ref
2 coforall box in cells[localDom] { // box
3   for nBox in neighs[box] { // neighbor box
4     for i in 1..box.count { // box atom
5       for j in 1..nBox.count { // nBox atom
6         if(dist(i, j) <= cutoff) {
7           fLocal.compute(i, j, fArr); // local
8 }}}}}
```

Code 7. Force loop with replication

We invoke `compute` on the local reference `fLocal` (lines 1 and 7) completely avoiding non-local access.

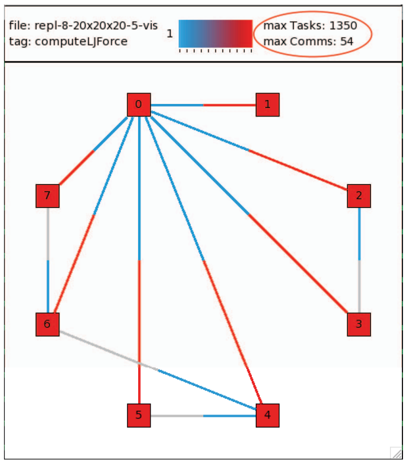


Figure 3. CoMD-Chapel - With force replication

If we run the same problem as above using this optimized implementation with replication we see from the chplvis output in Figure 3 that communication between locales during force computation has mostly been eliminated. The non-zero count is due to task creation by the master locale at the beginning of the force computation step (Section 3.2).

Although replication seems to be an obvious optimization, we discuss it here to emphasize the importance of programmer control on object locality. Accessing objects in a locale-agnostic manner can result in unintended communication. Tracking down offending accesses within the code is usually tricky because the language syntax and type system do not differentiate between local and remote objects. Moreover the Chapel runtime does not automatically cache remote objects. Its not clear if its possible to somehow enable this (compiling with the `--cache-remote` flag does not help in CoMD-Chapel.) Lastly, the performance penalty for unintentional remote access is significant. The implementation with force replication runs approx 1200x faster than the naive version (0.23s vs 280.97s.) Manual replication is clearly a necessary optimization for CoMD-Chapel.

4.2. Localization

Another aspect of the PGAS memory system is that besides normal pointers the code contains wide pointers that can refer to potentially remote data. Referring an object through a wide pointer even if it is local typically incurs higher runtime overhead.

Insufficient information may cause the compiler to generate wide pointers for otherwise completely local data. That is true with the optimized force loop in Code 7. Although the implementation clearly implies that `fLocal` on line 1 is a local object, the compiler still ends up generating wide pointers on line 7. In an attempt to avoid this problem, Chapel provides a mechanism to convey locality to the compiler.

4.2.1. `local` statement.

Code segments guaranteed to access only local data may be enclosed within a `local` statement. For instance, in Code 1, if the current locale and the new locale are guaranteed to be the same, then it can be rewritten as

```
1 var f:F = new F(); // create f
2 on (newLocale()) { // switch to new locale
3   ...
4   local { f.m(); } // f.m() is local
5   ...
6 }
```

Code 8. Localization

Here the compiler assumes that no non-local data access occurs on line 4 (and along every single code path reachable from it) and does not generate any wide pointers for that line. Given its MPI-like data decomposition, it is relatively easy to identify (but not refactor) local code segments in CoMD-Chapel. We show that code localization is another critical optimization for CoMD-Chapel.

We compare performance of the localized and non-localized versions of CoMD-Chapel. Figure 4 shows the difference in performance for a problem using 4x10^6 atoms and Figure 5 shows the same for a problem using 32000 atoms. Each simulation was run for 1000 timesteps on 1 locale and 8 locales. Both versions have force replication enabled.

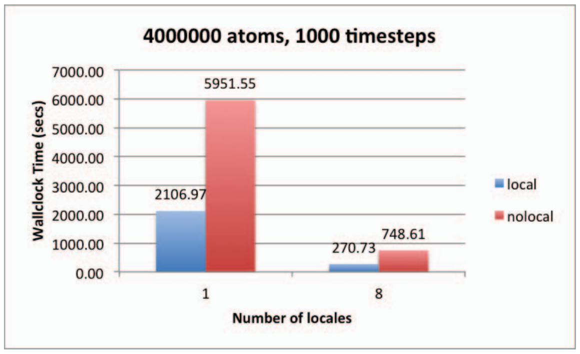


Figure 4. CoMD-Chapel - 4x10^6 atoms, local vs no local

Localizing wide pointers speeds up the code by about 3x. The impact is especially noticeable in the single-locale example which, despite being devoid of any inter-locale communication, slows down simply due to the use of wide pointers in the code (wide-pointer generation for a single-locale program can be completely avoided by compiling with the `--local` flag.) Clearly, wide pointers should be avoided whenever possible. To our knowledge, the `local` statement is the only Chapel construct that helps optimize wide-pointers. However, as we discuss next, it has limitations severely restricting its practical utility.

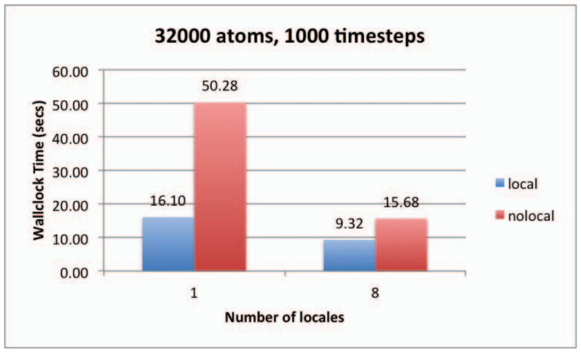


Figure 5. CoMD-Chapel - 32000 atoms, local vs no local

5. Type Based Locality

While the `local` statement is largely applicable to a small application like CoMD-Chapel, using it in more complex codes can be non-trivial.

To start with, the `local` statement is scope-based. This can inhibit localization without significant refactoring of the code. Lets reconsider our example from Code 8 with a few changes

```
var f:F = new F(); // create f
on (newLocale()) { // switch to new locale
  ...
  local { const x = f.m(); } // f.m() is local
  func(x); // use x
  ...
}
```

Code 9. Scoping issues with local statement

The above code will not compile since `x` is not visible outside the scope of the `local` statement. Making this work requires at least some code refactoring. In fact this results in several missed localization opportunities in CoMD-Chapel as well.

The `local` statement can also reject otherwise perfectly valid programs. Lets assume the method `m()` in Code 8 is defined as follows

```
1 class F {
2   proc m() {
3     on(Locales(1)) { // go to locale 1
4       func(); // local work on locale 1
5 }}}
```

Code 10. Implementation of method m()

Given this implementation of `m()`, Code 8 does not require any wide pointers. Yet, line 4 in Code 8 may fail since the method `m()` contains a task spawn to locale 1 (Code 10 line 3.)

A more serious concern is that the `local` statement asserts that every single data access within its block be completely local. It is the programmer's responsibility to ensure this holds for each line of code reachable from any call chain within the block. Not only is this tedious and error-prone, but it may not even be possible in instances where the code calls a function with hidden implementation details. For example, the only way to make Codes 8 and 10 work is to refactor lines 3-5 out of `m()`. If `class F` is implemented as a library, changing its implementation may not be feasible. Even if it is, the code still remains vulnerable to totally unrelated future changes.

Suppose we change line 1 in Code 8 as follows

```
var f:F = getF(); // factory method for F
```

If `getF()` does not guarantee the same locality properties as before, then suddenly line 4 isn't safe any more. Worse still the problem may not even be apparent until much later; the code will still compile, but may fail at runtime.

These problems restrict the overall usefulness of the `local` statement only to codes with visible and predictable access patterns; severely limiting its utility for complex applications.

It is instead more natural to treat locality as a property of data rather code blocks. An object's access pattern can be included in its type declaration.

```
var local f:F = getF();
```

This approach has several advantages over scope-based locality. First, only variables that are local should be marked as such. This allows programs like Codes 8 and 10 to run safely still using wide pointers where needed.

Secondly, making locality as a part of the type enables more precise compiler analysis. Type checking can disallow unsafe programs and generate compile-time warnings about potential wide pointer accesses, leading to better understanding and optimization of object accesses within the program.

Moreover, `local` statement assumes a strictly local/-global view of the address space. It is non-trivial to apply that to a more advanced memory model like hierarchical-PGAS [8], [9], [10] that extends traditional PGAS to a multi-level hierarchy of locales. Treating locality as a type can allow that through appropriate subtyping relationships.

```
local ≤ level1 ≤ level2 ≤ ... ≤ global
```

Lastly, extending the type system with data types having specialized semantics offers other possibilities e.g.,

```
var replicated force = new Force();
```

which provides automatic compiler-aided replication.

6. Performance

For performance evaluation, we consider the fully optimized version of CoMD-Chapel with both replication and localization enabled. The test platform is a cluster of nodes with *Intel Xeon EP X5660 2.8GHz* processor cores. Each node has 12 cores (single socket) and 24GB memory. The nodes are connected using an *InfiniBand QDR* high-speed interconnect. The Chapel compiler used is *v1.13.1* itself compiled with *gcc-4.9.2* and *ibv* as the *gasnet* conduit and *qthreads* as the threading framework. The code is compiled with the flags `--fast -snoRefCount`. Despite having 12 cores per node, we found that 10 threads per locale in our test setup gives the best performance. This may partly be due to a dispatcher thread used by the Chapel runtime.

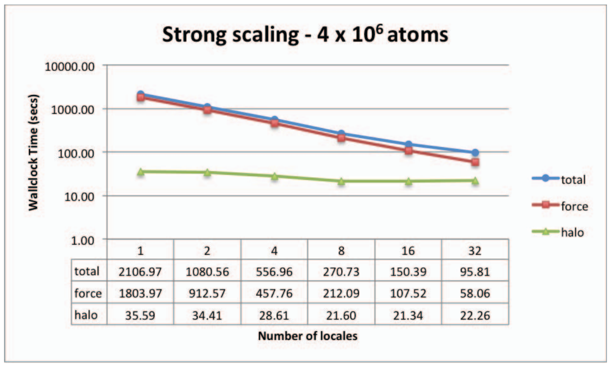
For each performance trend, we consider two problems sizes, *large* and *small*; the problem size determines the overall impact of Chapel runtime overheads. For each problem size we analyze the total execution time and time taken by the two heaviest steps—force and halo exchange.

All test cases are run for 1000 timesteps.

6.1. Strong scaling

We strong scale from 1 to 32 locales for a large problem of size 100^3 boxes (= $4\text{x}10^6$ atoms) and a small problem of size 20^3 boxes (= 32000 atoms).

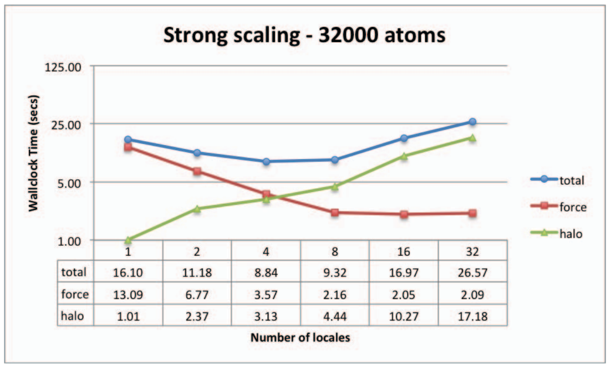
Consider the performance of the larger problem as shown in Figure 6. As expected the force step scales quite well. Characterization of the halo exchange step is more difficult. First, it initiates three times the number of remote task launches (Code 5, line 4) as the other steps (Code 3, line 3). Thus any task creation overheads are much more pronounced for halo exchange. Furthermore, in simulations with less than 8 locales, at least part of the communication between neighbors is local. Thus additional analysis is required to understand the observed scaling behavior of halo exchange step. However as long as the dominant step (force) scales well, the overall problem scales well too.



	1	2	4	8	16	32
total	2106.97	1080.56	556.96	270.73	150.39	95.81
force	1803.97	912.57	457.76	212.09	107.52	58.06
halo	35.59	34.41	28.61	21.60	21.34	22.26

Figure 6. Strong scaling 4x10^6 atoms

For the small problem (Figure 7) the force step scales well up to 8 locales. However, since the contribution of the force step to the overall execution time is less significant than the halo exchange the overall problem does not scale well.



	1	2	4	8	16	32
total	16.10	11.18	8.84	9.32	16.97	26.57
force	13.09	6.77	3.57	2.16	2.05	2.09
halo	1.01	2.37	3.13	4.44	10.27	17.18

Figure 7. Strong scaling 32000 atoms

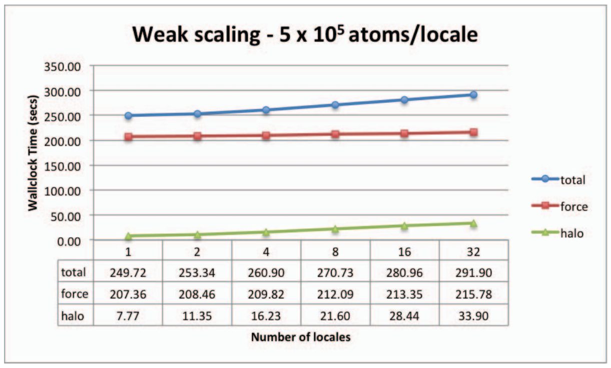
6.2. Weak scaling

For weak scaling, we select a large problem of size 50^3 boxes/locale (= 5x10^5 atoms/locale) and a small problem of size 10^3 boxes/locale (= 4000 atoms/locale). We run this configuration on 1 to 32 locales. Again the large problem (Figure 8) shows better scaling than the small problem (Figure 9) for reasons stated above.

However, all the steps (including force) show some performance degradation that roughly scales with the number of locales. This again points to possible overheads associated with remote task creation (Code 3, line 3 and Code 5, line 4.)

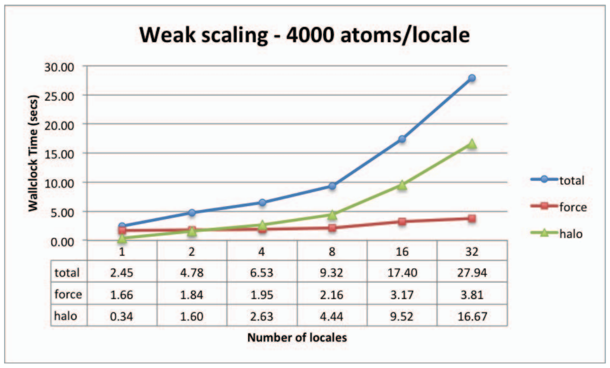
6.3. Comparison to CoMD

We compare the performance of fully-optimized version of CoMD-Chapel with the reference implementation on 8 and 32 locales. We use a problem of size 100^3 boxes (= 4x10^6 atoms).



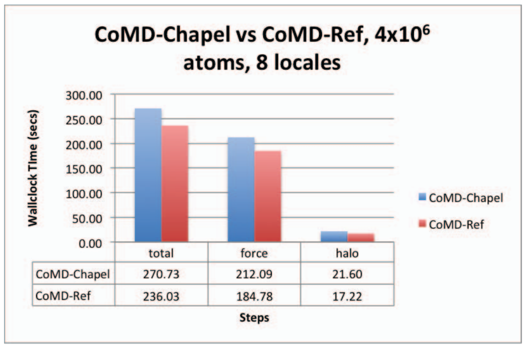
	1	2	4	8	16	32
total	249.72	253.34	260.90	270.73	280.96	291.90
force	207.36	208.46	209.82	212.09	213.35	215.78
halo	7.77	11.35	16.23	21.60	28.44	33.90

Figure 8. Weak scaling 5x10^5 atoms/locale



	1	2	4	8	16	32
total	2.45	4.78	6.53	9.32	17.40	27.94
force	1.66	1.84	1.95	2.16	3.17	3.81
halo	0.34	1.60	2.63	4.44	9.52	16.67

Figure 9. Weak scaling 4000 atoms/locale



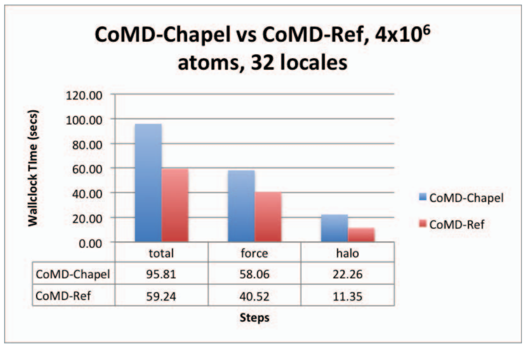
	total	force	halo
CoMD-Chapel	270.73	212.09	21.60
CoMD-Ref	236.03	184.78	17.22

Figure 10. CoMD-Chapel vs CoMD (8 locales)

On 8 locales (Figure 10), CoMD-Chapel performs to within 87% of the reference implementation. However that drops to only 62% of the reference implementation on 32 locales (Figure 11). On 8 and 32 locales, the force computation step performs respectively up to 87% and 70% of the reference implementation. This drop in performance is due to an increasing remote task creation overhead for larger locale counts.

7. Related Work

Johnson et al. [11] have looked into optimizing single-locale Chapel performance. We seek to optimize the multi-



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

Figure 11. CoMD-Chapel vs CoMD (32 locales)

locale case. miniMD [12] is another multi-locale molecular dynamics proxy written in Chapel. Our strategy for data distribution across locales is very similar to the SPMD version of miniMD although the application logic is completely different. Moreover, whereas miniMD primarily analyzes a stencil abstraction for data decomposition, we are more interested in evaluating the Chapel language support for enabling multi-locale optimizations. Chapel features useful for improved PGAS access have also been explored by Kayraklioglu et al. [13]. We further analyze limitations of these and discuss the need for advanced capabilities. Murata et al. [14] have studied the scaling properties of CoMD in the X10 programming language [15], [16] for a multiple place (locale) implementation.

8. Conclusions and Future Work

We have implemented a multi-locale Chapel version of the CoMD benchmark. We show that object access optimizations like replication and localization are critical to achieve performance comparable to the reference implementation. We also argue that existing scope-based locality has limited usability beyond simple codes and that type-based locality is a more desirable approach.

For future work, we want to explore ways to optimize the Chapel runtime overheads especially the cost of task creation from the master locale for each step (Section 3.2). We intend to extend this work by abstracting data decomposition details from application logic using stencil distribution and parallel zippered iterators [17] and finally evaluate the performance of that implementation. Finally, we want to further expand upon the notion of type-based object locality.

Acknowledgments

We would like to thank Tom MacDonald, Ben Albrecht and Ben Harshbarger from the Chapel development team for their invaluable advice and timely help in optimizing the various aspects of the CoMD-Chapel code.
CoMD was developed by the ExMatEx co-design center.
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 and supported by Office of Science, Office of Advanced Scientific Computing Research.

References

[1] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[2] B.L. Chamberlain. Chapel. In Pavan Balaji, editor, *Programming Models for Parallel Computing*, chapter 6, pages 129–159. The MIT Press, 2015.

[3] Chapel Language Specification. http://chapel.cray.com/docs/1.13/_downloads/chapelLanguageSpec.pdf.

[4] Co-design for Molecular Dynamics (CoMD). http://www.exmatex.org/comd.html.

[5] A. Sanz, R. Asenjo, J. Lpez, R. Larrosa, A. Navarro, V. Litvinov, S. E. Choi, and B. L. Chamberlain. Global data re-allocation via communication aggregation in chapel. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 235–242, Oct 2012.

[6] CoMD github. https://github.com/exmatex/CoMD.

[7] chplvis : Visualization tool for Chapel. http://chapel.cray.com/docs/latest/tools/chplvis/chplvis.html.

[8] Chapel Hierarchical Locales. http://chapel.cray.com/presentations/SC14/ChapelLocaleModels.pdf.

[9] Amir Kamil and Katherine A. Yelick. Hierarchical computation in the SPMD programming model. In *Languages and Compilers for Parallel Computing - 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25-27, 2013. Revised Selected Papers*, pages 3–19, 2013.

[10] Amir Kamil and Katherine A. Yelick. Hierarchical pointer analysis for distributed programs. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pages 281–297, 2007.

[11] R. B. Johnson and J. Hollingsworth. Optimizing chapel for single-node environments. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1558–1567, May 2016.

[12] miniMD in Chapel. http://chapel.cray.com/presentations/SC13/01-minimd-chamberlain.pdf.

[13] PGAS Access Overhead Characterization in Chapel. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7530053.

[14] Porting MPI based HPC Applications to X10. x10.sourceforge.net/documentation/papers/X10Workshop2014/x1014-murata.pdf.

[15] X10 Language Specification. http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.

[16] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[17] User-Defined Parallel Zippered Iterators in Chapel. pgas11.rice.edu/papers/ChamberlainEtAl-Chapel-Iterators-PGAS11.pdf.