Efficient update of ghost regions using active messages

Josh Milthorpe
Alistair P. Rendell
Research School of Computer Science
Australian National University
{josh.milthorpe, alistair.rendell}@anu.edu.au

***Abstract*—The use of ghost regions is a common feature of many distributed grid applications. A ghost region holds local read-only copies of remotely-held boundary data which are exchanged and cached many times over the course of a computation.**

X10 is a modern parallel programming language intended to support productive development of distributed applications. X10 supports the "active message" paradigm, which combines data transfer and computation in one-sided communications. A central feature of X10 is the *distributed array*, which distributes array data across multiple places, providing standard read and write operations as well as powerful high-level operations.

We used active messages to implement ghost region updates for X10 distributed arrays using two different update algorithms. Our implementation exploits multiple levels of parallelism and avoids global synchronization; it also supports split-phase ghost updates, which allows for overlapping computation and communication. We compare the performance of these algorithms on two platforms: an Intel x86-64 cluster over QDR InfiniBand, and a Blue Gene/P system, using both stand-alone benchmarks and an example computational chemistry application code. Our results suggest that on a dynamically threaded architecture, a ghost region update using only pairwise synchronization exhibits superior scaling to an update that uses global collective synchronization.

***Index Terms*—parallel programming models; Partitioned Global Address Space (PGAS); X10 language; active messages; structured grids; distributed arrays; ghost regions; lattice Boltzmann method; Smooth Particle Mesh Ewald method**

I. Introduction

Spatial grids[1] are used in a wide range of scientific application codes including computational fluid dynamics, weather simulations and molecular dynamics. Computation on each grid element requires only those elements in the immediate neighborhood – a locality property that may be exploited in distributed algorithms. A common feature of many distributed grid codes is the use of ghost regions. A ghost region holds read-only copies of remotely-held data, which are cached to enable local computation on boundary elements. Grid applications are typically iterative, so processes coordinate to exchange and cache ghost data many times over the course of a computation. The efficient implementation of ghost region updates is therefore an important factor in achieving performance and scalability.

[1]or meshes, not to be confused with grid computing architectures

Traditionally, grid applications for distributed memory architectures have been implemented using the message passing programming model, exemplified by MPI[1]. However, the Partitioned Global Address Space (PGAS) programming model is becoming increasingly popular. The PGAS model provides a global view of memory in which a portion of the memory is local to each process, making it straightforward to write programs in which processes access remote data, while still accounting for locality and associated communication costs.

Modern PGAS languages such as X10[2] and Chapel[3] support the *active message* idiom, which combines data transfer and remote computation within a single message; this allows for the terse expression of many distributed algorithms using localized patterns of communication and synchronization. X10 is a PGAS language which explicitly represents locality in the form of places. A *place* in X10 corresponds to one or more co-located processing elements with attached local storage. Each place supports a set of dynamically spawned lightweight *activities*, spawned using the *async* statement. An activity runs to completion at the place where it was started, and may spawn new remote activities (*at*) and detect termination of spawned activities (*finish*). Mutual exclusion is ensured between activities running at the same place by the use of conditional and unconditional atomic sections (*when* and *atomic*). X10 also provides a rich array sub-language. An *array* is a collection of objects which are indexed by *points* in an arbitrary bounded *region*. A key data structure in X10 is the *distributed array*. Each element in a distributed array is assigned to a particular place according to the array *distribution*, which is a mapping from point to place[2]. Distributed arrays are very general in scope: they allow for arbitrary distributions (for example, block, block-cyclic, recursive bisection, fractal curve) and arbitrary regions (for example dense or sparse, rectangular, polyhedral or irregular). The X10 language specification, programmer's guide and tutorials are available at http://x10-lang.org.

The Global Arrays library provides PGAS support for array data, including ghost region updates, and has been used successfully to develop structured grid applications as well as a wide range of other codes[4]. Global Arrays supports one-sided put and get operations, synchronization and collective operations but does not support active messages. Palmer and

Nieplocha[5] compared algorithms for the update of ghost regions for block-distributed rectangular arrays for the Global Arrays library. They identified two basic algorithms: a *put* algorithm in which each process sends ghost data to all neighboring processes, and a *shift* algorithm which reduces the number of messages required by making use of the fact that some ghost data are required by more than one process. They compared implementations of these algorithms on two different architectures – the IBM SP and the Cray T3E – and found that different algorithms were more efficient on each architecture depending on the relative cost of communication and synchronization.

The PETSc[6] library also provides support for distributed array ('vector') data with ghost regions. Data transfer and synchronization are implemented using MPI functions.

This paper describes how active messages can be used to implement ghost region updates as a high-level operation on X10 distributed arrays. The implementation uses only local synchronization, which results in improved scaling to larger numbers of distributed processors. It also supports split-phase updates, allowing communication of ghost data to be overlapped with computation at each place.

Section II describes the X10 array library in detail. Section III presents two well-known algorithms for ghost region updates and our implementation of these algorithms in X10. In section IV, we follow and extend the work of Palmer and Nieplocha[5] with a comparison of the two algorithms on two modern systems: a cluster of Intel Nehalem nodes with an InfiniBand interconnect, and a Blue Gene/P system. We demonstrate that the total number of messages required to transfer ghost data is of secondary importance compared to the method of synchronization. We also identify a limitation of the current X10 compiler that makes it difficult to efficiently implement high-level operations for generic array types.

II. Distributed arrays in X10

The X10 standard class library provides a rich set of array classes in the `x10.array` package[7]. As X10 supports a constrained type system[8], array types can carry various constraints with regard to dimensionality, shape and range properties. We describe the array classes here as background for our implementation of ghost regions in distributed arrays in Section III.

The fundamental X10 array classes are `Point`, `Region` and `Array`. A `Point(n)` is an index of rank (dimensionality) n, for example `[1,0]` is a `Point(2)`. A `Region(n)` is a set of `Points` of rank n. A `Region` may be of any shape (for example, rectangular, triangular, general polyhedral, sparse); the only requirement is that it must be possible to iterate over all points in the region. An `Array` is a local collection of data indexed by `Point`, with one element corresponding to each `Point` in a `Region`.

Distribution of array data is supported by the classes `PlaceGroup`, `Dist` and `DistArray`. A `PlaceGroup` is a set of `Places`. One static instance of this class, `PlaceGroup.WORLD`, is defined as the set of all places involved in the computation. A `Dist` (distribution) divides a `Region` between the places in a `PlaceGroup`, by mapping each `Point` in the `Region` to a single `Place`. A `DistArray` is a collection of indexed data distributed over places according to a given `Dist`.

Both `Array` and `DistArray` support standard subscripting operations to get and set individual elements. They also support higher-level operations such as:

- `sequence()` - iterate over all elements in sequence
- `fill()` - set every element to a defined value
- `map(op:(T)=>U)` - return a new array that is the result of applying the given map function to each element
- `reduce(op:(T,T)=>T, unit:T)` - reduce all elements in the array using the given reduction function e.g. sum, min, max

These high level operations may be implemented using efficient parallel algorithms, which makes them useful building blocks for high performance application codes. Our implementation of ghost region updates complements these high level operations.

Array operations are optimized for the specific case of dense, rectangular arrays. A small number of standard distributions are supplied as subclasses of `Dist`. However, these are implemented entirely in X10 code (without special compiler support), so there is nothing to prevent users creating their own subclasses of `Dist` in a similar manner.

Although ghost regions are useful in both structured and unstructured grid computations, for the purpose of this paper we limit our discussion to structured grid applications in which each place holds a portion of a larger dense, rectangular grid. For structured grid applications, the distributions of most interest are:

- `BlockDist` - a regular distribution of a rectangular region, divided along one dimension
- `BlockBlockDist` - a regular distribution of a rectangular region, divided along two dimensions

We modified the X10 array library to support ghost region updates for both of these distributions.

III. Implementing ghost regions in X10 distributed arrays

We implemented ghost regions as new classes in the package `x10.array`, as well as modifications to the existing `DistArray` class.

We added a method, `Region.getHalo(haloWidth:Int)`, which returns a halo region comprising the neighborhood of the target region. For rectangular regions, the halo region is simply a larger rectangular region enclosing the target region. For the special case of a zero-width ghost region (no ghosts), `Region.getHalo(0)` returns the region itself.

X10 makes use of a natively implemented generic class, `PlaceLocalHandle`, to allocate and track a unique object at each place. `DistArray` is implemented using a `PlaceLocalHandle[DistArray.LocalState]`, which holds local data for each place. In the constructor for

`DistArray`, the `Dist` is used at each place to determine the region that is mapped to that place (the *resident* region), and the local state is allocated to hold the portion of the distributed array corresponding to that region.

We changed the constructor to allocate storage for the ghost region in `LocalState`, and added a new field `LocalState.ghostManager:GhostManager` which holds a distribution-specific object that manages ghost updates. This reduces to "standard" behavior for `DistArray` in the special case of `ghostWidth==0` as there is no ghost manager and the ghost region is identical to the resident region. All operations on `DistArray` were changed to use the ghost region rather than the resident region for indexing.

The implementation of the `GhostManager` interface is specific to the distribution type. It may also be implemented using different algorithms depending on the target architecture. We implemented the two basic algorithms described by Palmer and Nieplocha[5] (see §III-E and §III-F).

A. Library interface

The following methods are defined on `DistArray` and constitute the user API for our implementation:

- `sendGhostsLocal()`
  a single-place operation that sends boundary data from this place to the ghost regions stored at neighboring places
- `waitForGhostsLocal()`
  a single-place operation that waits for ghost data at this place to be received from all neighboring places
- `updateGhosts()`
  a global operation that is called at a single place to update ghost regions for the entire array; this starts an activity at each place in the distribution to send and wait for ghosts

B. Synchronization requirements

Ghost region updates are typically used in the context of a phased, iterative computation, for example:

```
for (i in 1..ITERS) {
  updateGhosts();
  computeOnLocalData();
  computeOnGhostData();
}
```

It is necessary to synchronize between reads and updates to ensure that ghost regions have been fully received before computation begins at each place.

There are two basic approaches to this problem. One is to use implicit synchronization through two-sided (send/receive or scatter/gather) communications. This is the approach used in the PETSc[6] library, and in the M_P (message-passing) algorithm described by Palmer and Nieplocha[5].

An alternative is to use one-sided communications surrounded by explicit synchronization. In some computations, such synchronization may naturally be included in the computation. In the following example, an energy and a maximum change per grid point are calculated at different times in a structured grid code:

```
for (i in 1..ITERS) {
  energy = allReduce(energy, SUM, ...);
  computeOnLocalData();
  sendGhosts();
  maxDelta = allReduce(maxDelta, MAX, ...);
  computeOnGhostData();
}
```

The collective reduction operations surrounding the ghost update ensure the consistency of ghost data by enforcing an ordering with other messages. All previous send operations from a place must complete before the collective reduction can begin. Where such natural synchronization is not present, the ghost update operation must perform synchronization before and after sending ghosts. In Global Arrays this synchronization is done with a global collective operation.

Our approach combines non-blocking messages with local synchronization as suggested by Kjolstad and Snir[9]. We assign a phase counter to each ghosted `DistArray`. The use of a unique phase counter per array allows ghost updates on different arrays to proceed independently. This could be of use for example in a multigrid or adaptive mesh refinement algorithm in which different timesteps are used for coarser or finer grids. In each even-numbered phase the program computes on ghost data; in each odd-numbered phase ghost data are exchanged with neighboring places. A place may not advance more than one phase ahead of any neighboring place. A call to `sendGhostsLocal()` increments the phase for this place and then sends active messages to update ghost data at neighboring places. Each active message also sets a flag to notify the receiving place that data have arrived from a particular neighbor. The receiving place calls `waitForGhostsLocal()` to check that flags have been set for all neighbors before proceeding with the next computation phase.

C. Split-phase updates

The use of local synchronization allows phases to proceed with computation before neighboring places have received their ghost data; it also allows communication of ghost data to overlap with computation on local data at each place, as follows:

```
// at each place
for (i in 1..ITERS) {
  sendGhostsLocal();
  computeOnLocalData();
  waitForGhostsLocal();
  computeOnGhostData();
}
```

D. Use of active messages

An active message both transfers data and initiates computation at a remote place; in X10, this computation may include synchronization with other activities running at the target place. For example, in the following code, activity 1 sends an active message to fill a buffer at place `p` with a value

of type `T`. Activity 2 running at place `p` waits for the buffer to be filled, and then removes and computes on the value.

```
// activity 1
val v:T = ...;
async at(p) {
  when(!buffer.full) {
    buffer.put(v);
  }
}

// activity 2 at Place p
async {
  val v:T;
  when(buffer.full) {
    v = buffer.remove();
  }
  computeOn(v);
}
```

In our implementation of ghost updates, active messages are used to transfer and perform local layout of ghost data, and to ensure consistency of data for each phase of computation. In the shift algorithm (§III-F) additional synchronization is required between the two phases of the shifts.

In `sendGhostsLocal()`, each place sends messages to neighboring places. A conditional statement (`when`) ensures that the ghost data are not updated until the receiving place has entered the appropriate phase. After ghost data have been updated, a flag is set within an atomic section to indicate that the data have been received:

```
at(neighbor) async {
  val mgr = localHandle().ghostManager;
  when (mgr.currentPhase() == phase);
  for (p in overlap) {
    ghostData(p) = neighborData(p);
  }
  atomic
    mgr.setNeighborReceived(sourcePlace);
}
```

In `waitForGhostsLocal()`, another conditional atomic block is used to wait until ghost data have been received from all neighboring places:

```
public def waitForGhostsLocal() {
  when (allNeighborsReceived()) {
    currentPhase++;
    resetNeighborsReceived();
  }
}
```

E. Ghost updates using the put algorithm

The *put* algorithm is the most straightforward way to transfer ghost data between places[5].

At each place:

1) determine the list of neighboring places
2) for each neighbor:
   a) determine the overlap between data held at this place and the neighbor's ghost region
   b) send the overlapping data to the neighbor and store in the neighbor's ghost region.

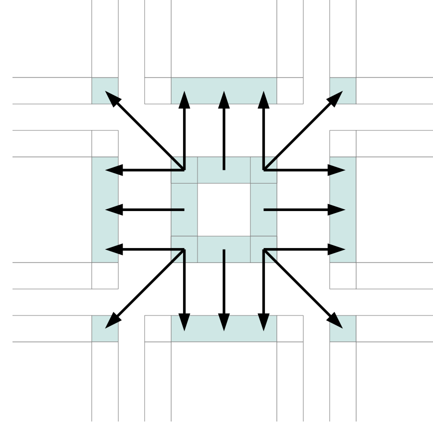

Fig. 1: Ghost update *put* algorithm in two dimensions: ghost data are sent from a place to all neighboring places. Note: exchange of data between other places is not shown.

This algorithm requires a maximum of $3^D - 1$ messages per place, where D is the number of divided dimensions in the distribution. For example, in a block dist divided in one dimension, there is a maximum of 2 messages per place, and in a two-dimensional block/block dist (see figure 1), there is a maximum of 8 messages per place. No ordering or synchronization is required between the messages to each neighbor.

F. Ghost updates using the shift algorithm

The *shift* algorithm makes use of the fact that after some of the ghost data have been received at a place, some of those data are required by other neighbors and can be passed on in messages to those neighbors[5].

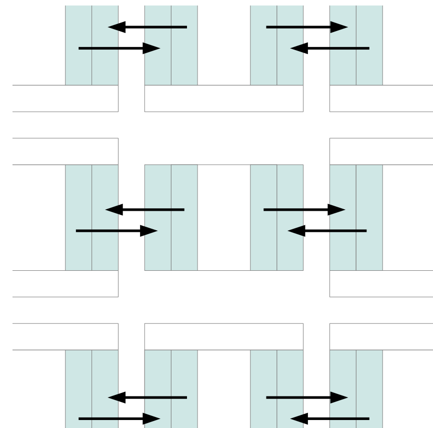

Fig. 2: Ghost update *shift* algorithm in two dimensions: Step 1: ghost data are sent from each place to neighboring places along one axis.

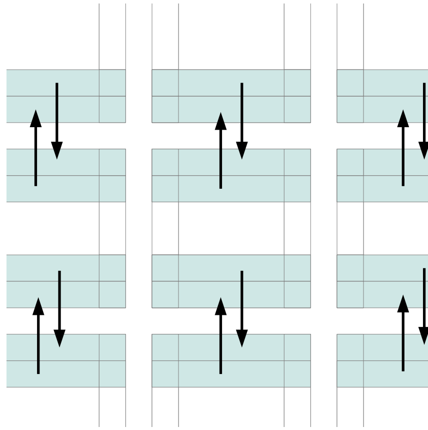

Fig. 3: Ghost update *shift* algorithm: Step 2: ghost data are sent to neighboring places along the other axis, including those data previously exchanged.

At each place:

1) for each axis d in the distribution:
   a) determine the neighboring places along the axis d
   b) for each neighbor:
      i) determine the overlap between data held at this place – including ghost data already received from other places – and the neighbor's ghost region
      ii) send the overlapping data to the neighbor and store in the neighbor's ghost region.

This reduces the number of required messages to a maximum of $2D$ where D is the number of divided dimensions, at a cost of increased synchronization.

For example, in a block/block dist, a maximum of 4 messages per place is required (see figures 2 and 3). However, an additional synchronization step is required. Each place must wait until ghost data have been received from places along one axis before proceeding to send data along the next axis.

IV. Evaluation

We evaluated the performance of our ghost update implementation using microbenchmarks and in the context of complete application codes. Evaluation was performed on two machines: the *Watson 4P* Blue Gene/P system at IBM Watson Research Center, and *Vayu*, which is an Oracle/Sun Constellation cluster installed at the NCI National Facility at the Australian National University. Each node of *Vayu* is a Sun X6275 blade, containing two quad-core 2.93GHz Intel Nehalem CPUs, 24GB DDR3-1333 memory and on-board QDR InfiniBand.

For all reported results, the Native (C++ backend) version of X10 2.2.2.1 was used. Single-threaded places were used (`X10_NTHREADS=1`) to avoid complications from interactions with the work-stealing runtime. Therefore 8 places were run on each dual quad-core node of *Vayu*, and 4 places on each quad-core node of *Watson 4P*.

Version 5.1 of Global Arrays was used for comparison (see §IV-C).

A. Alternative algorithm for static threads

The default implementation of the X10 runtime uses a dynamic thread pool for each place. At the start of computation, the number of threads created at each place is determined by the environment variable `X10_NTHREADS`. Each thread maintains a double-ended queue (deque) of activities to be executed. Whenever an activity is created, it is pushed to the bottom of the deque of the creator thread. An idle thread may steal work from other threads by removing activities from the tops of their deques[10].

When an activity must block — for example waiting for the condition guard on an atomic block to evaluate to `true` — the X10 runtime "parks" the worker thread that is executing that activity, and creates an additional thread to process other queued activities. This ensures that the same level of parallelism is maintained throughout a computation, even in the presence of blocking constructs. In other words, at any time there are at most `X10_NTHREADS` active threads, although there may be additional threads that are blocked.

However, some architectures do not support dynamic thread creation; this includes Blue Gene/P. To support these architectures, dynamic thread creation may be disabled by setting the environment variable `X10_STATIC_THREADS`. With the current implementation of `X10_STATIC_THREADS`, it is possible for otherwise correct application code using blocking constructs to deadlock, as all threads may be blocked waiting on activities that are further back in the work queues.

Due to the above limitation, our implementation of ghost region update worked correctly for any number of threads on the dynamically-threaded *Vayu* machine, but deadlocked on the statically-threaded *Watson 4P*. As a workaround, we implemented a version of the ghost update using collective barrier synchronization, similar to that in Global Arrays; this is the version used with `X10_STATIC_THREADS=true` on Blue Gene (*Watson 4P*).

B. Ghost update microbenchmark results

Figure 4 presents ghost update timings with the *put* and *shift* algorithms on *Watson 4P* and *Vayu*. This benchmark performs 10,000 ghost updates sequentially over a large distributed array. The array size for one place is 100^3, and the array size is increased with the number of places (weak scaling), so that each place always holds one million double-precision values.

The time for 2 and 4 places are significantly lower than for 8 or more places. This is expected for two reasons: when there are less than 16 places, there are no interior places with neighbors on all sides; and as both platforms support 4 places on a single CPU, all communication is done through shared memory thereby avoiding the network. Scaling is fairly flat above 8 places, as the number and size of messages sent by each place is roughly constant.

On *Watson 4P* (Blue Gene/P), the put algorithm is significantly faster. On *Vayu* (Nehalem + InfiniBand), the shift

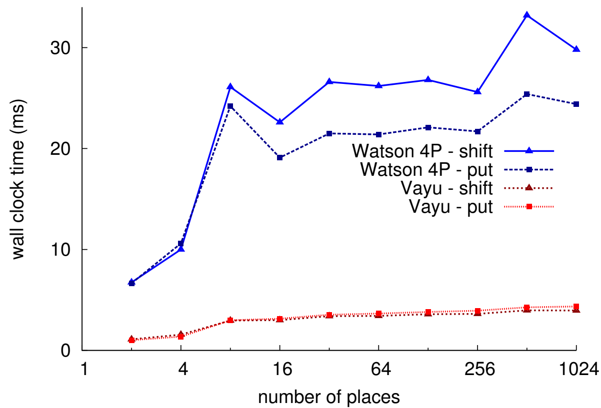

Fig. 4: Ghost region update: weak scaling for three-dimensional array using *put* and *shift* algorithms on *Vayu* (Nehalem + InfiniBand) and *Watson 4P* (Blue Gene/P).

algorithm is marginally faster. This reflects the relative balance between CPU and network performance on the two architectures; the reduced number of messages for the shift algorithm is more than outweighed on Blue Gene/P by the cost of synchronizing X10 activities. This is consistent with Palmer and Nieplocha's finding that synchronization costs significantly affect the relative performance of the algorithms depending on architectural characteristics[5].

Figure 5 demonstrates the benefit of using only local synchronization as compared to a global barrier. We performed the same benchmark on *Vayu* to compare our local-synchronization implementation of the shift algorithm with the alternative algorithm using a global barrier (§IV-A).

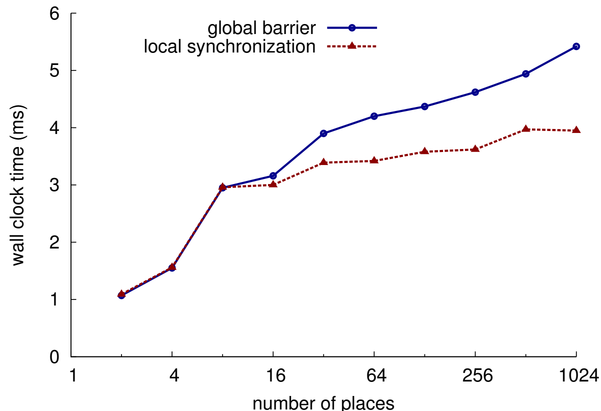

Fig. 5: Ghost region update: weak scaling with global barrier vs. local synchronization on *Vayu*.

For 2-8 places, the difference between local and global synchronization is insignificant, as all communications (including the global barrier) take place within a single node. For more than eight places, the update time with a global barrier increases more rapidly than the time with only local synchronization. This is expected as the collective operation on which it is implemented scales as $0(\log p)$ where p is the number of places, whereas other elements of the ghost update scale as $0(1)$.

These results suggest that the use of local vs. global synchronization is more important to overall performance than the difference in synchronization requirements due to the update algorithm (put vs. shift).

C. Application: Lattice Boltzmann

The Global Arrays library[4] is distributed with a sample Fortran code that performs a two-dimensional lattice Boltzmann simulation of flow in a lid-driven square cavity. This has been used to demonstrate GA's ghost update capabilities[5]. We implemented an equivalent code in X10 to enable direct comparison between our ghost update implementation and that in GA. The X10 code is significantly shorter: around 600 (non-comment) lines of code compared to almost 900 in the Fortran/GA example. (The X10 application also uses less memory in execution, at about 94MB per process compared to 156MB per process with Fortran/GA. The difference in memory footprint is probably due to maintaining local and globally shared copies of data in Global Arrays.)

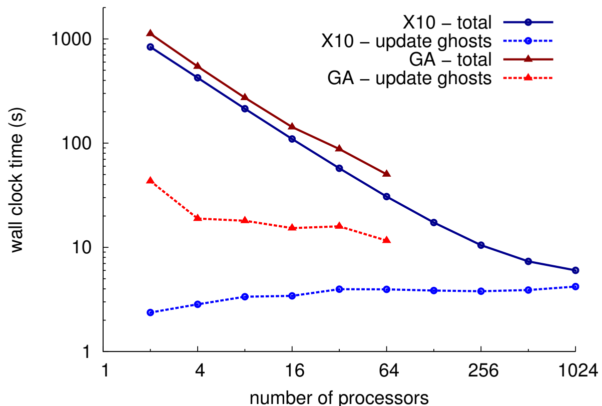

Fig. 6: Timing comparison between X10 and Fortran / Global Arrays code for a 5000 step lattice Boltzmann simulation on 1024 × 1024 grid on *Vayu*. Both the total time for the calculation and the ghost region update time are shown.

Figure 6 presents strong scaling results for both the X10 and Fortran/GA codes on *Vayu*. The X10 code exhibits superior performance and scaling, and is almost 40% faster on 64 places (31.1s total compared to 50.3s for the Fortran/GA code).[2] The scaling of the X10 code starts to reduce noticeably above 256 places. For this number of places, more than one third of the computation time is spent in updating ghost regions. As the time for ghost updates is roughly constant regardless of the number of places, the update of ghost regions becomes the dominant factor in scaling when the problem is divided

[2]The Global Arrays example code currently crashes with a segmentation violation for ≥128 processes for this problem size.

finely enough. It would be necessary to increase the problem size to achieve further (weak) scaling.

D. Application: Particle Mesh Ewald

Our second application example is from the chemistry domain: a Smooth Particle Mesh Ewald (PME) electrostatics calculation from the ANUChem suite of chemistry codes in X10[11]. The code uses lattice-centric charge interpolation as suggested by Ganesan et al.[12], but differs from their scheme in that it does not use neighbor lists. Instead, the code divides the charges into subcells with a side length equal to half the direct interaction cutoff distance. Each place considers charges in a region of subcells surrounding its resident lattice points. The subcells are also used in the calculation of direct particle-particle interactions. Ghost regions are used to store a halo of neighboring subcells for each place, to support efficient local calculation of the direct interaction.

The previously published version of the code used an application-specific ghost region update to gather subcells from neighboring places. We altered the code to use our ghost cell update algorithm, which reduced the size of the application code from 682 to 617 non-comment source lines. Figure 7 presents strong scaling results for a system of 17,132 water molecules.

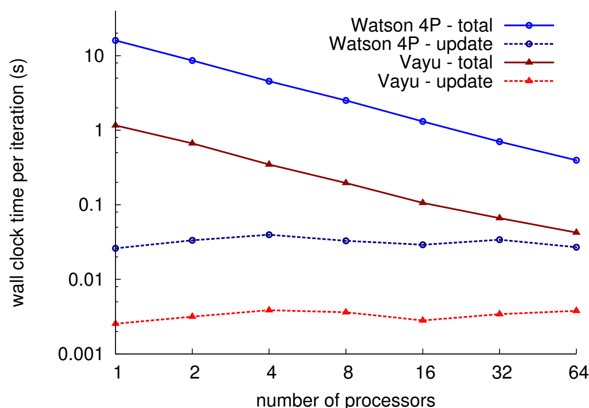

Fig. 7: ANUChem PME electrostatics calculation for 17K water molecules on *Vayu* (Nehalem + InfiniBand) and *Watson 4P* (Blue Gene/P). Both the total time for the calculation and the subcell ghost region update time are shown.

The modified PME code scales well from 1-64 places on both *Vayu* and *Watson 4P*. We were not able to gather timings on more than 64 places because the width of the ghost region for this application is too small (see §VI-A2).

V. Related Work

The Global Arrays library (GA)[4] supports distributed array functionality, including ghost cells and periodic boundary conditions, in the context of C++ and Fortran applications. Palmer and Nieplocha[5] evaluated alternative algorithms for GA ghost cell updates on the Cray T3E and IBM SP architectures. This paper updates their work for current architectures (Blue Gene/P and Nehalem+InfiniBand), and reevaluates the two basic algorithms (shift and put) in the context of the PGAS programming model with active messages.

PETSc[6] also supports distributed arrays with ghost data and periodic boundary conditions. If ghost data are required, the storage vector must be duplicated into an extended "local" vector with room for the ghost data. PETSc uses blocking scatter and gather operations to exchange ghost data between processes.

Wen et al.[13] implemented exchange of irregular ghost regions in Titanium (a PGAS language based on Java) for an Adaptive Mesh Refinement application. Their implementation overlaps computation and communication and uses global synchronization. They also suggest that their approach may transfer to X10, but no implementation work has been undertaken.

Claridge[14] also implemented exchange of irregular ghost regions in Chapel (a PGAS language similar to X10) for an Adaptive Mesh Refinement library.

VI. Discussion and Future Work

Our experience developing the ghost update implementation suggests a number of possible improvements. We divide these improvements into two categories: those related to the update algorithm, and those related to language support for implementing the algorithms in X10.

A. Algorithmic improvements

1) Irregular distributions: We implemented ghost regions for 1- and 2-dimensional block distributions (see section II), which are most relevant to structured grid applications. Other distributions may also be of interest including Morton (Z-index) and irregular distributions.

2) Fallback algorithm for extended ghost regions: Ghost regions for structured grid calculations are often a single cell in width, as only immediately neighboring data are needed to calculate at each point. However, it may be advantageous to exchange wider ghost regions, to allow less frequent updates[15].

If the ghost region is wide enough and the local region held at each place narrow enough in a particular dimension, it is possible that the ghost region for a place will extend beyond its immediate neighbors. In this case, the fast *put* and *shift* algorithms described above will not work, and it is necessary to fall back to a slower, more general algorithm that calculates ghost region overlaps for all places rather than for the immediate neighbors only. This is the approach used in the Global Arrays library[4]. We have not yet implemented this fallback approach in X10. Such extended ghost regions are not likely to occur for typical problem sizes in structured grid calculations, however we plan to implement such a fallback algorithm to complement the fast algorithms described above, to ensure that the implementation is robust in this case.

B. Language implementation issues

1) Atomic blocks: Our local synchronization algorithm is implemented in the X10 language using atomic blocks. The current implementation of atomic blocks in X10 uses a place-wide lock. While semantically correct, this choice means that at any given time, only one thread may progress in an atomic block. For single-threaded places, this does not introduce additional overhead. However, for multithreaded places, the time that threads spend waiting on the place-wide lock may become significant.

X10 allows for a number of different implementations of atomic semantics, including finer-grained locks and software transactional memory[2], which would improve the performance of local-synchronization ghost updates, as threads could handle incoming active messages with atomic blocks in parallel.

2) Inlining of Point objects: The implementation of both *put* and *shift* algorithms was rendered more difficult by a current limitation of the X10 compiler. Where the rank of an array is statically known, it may be indexed using standard integer indices e.g. `a(4,2)`. However when the rank is a dynamic constraint — as is typical in library code — arrays must be indexed using `Point` e.g. `a(p)`. Objects of type `Point` (which may be of arbitrary rank) are not currently inlined. This means that array indexing using `Point` generates a high overhead of `Point` object creation and method calls. To measure the overhead due to the use of `Point` objects, a "hand-inlined" version using only integer indices was constructed for both algorithms for `DistArray(3)`. Table I shows the speedup of the "hand-inlined" versions relative to the `Point` versions of the code, on 16 places on *Vayu*. These results indicate that `Point` inlining would significantly improve the performance of ghost updates for arrays of arbitrary rank, and may be expected to improve the performance of similar algorithms. Hand-inlined versions of both algorithms were used for all benchmarks reported above.

TABLE I: Speedup from "hand-inlining" of `Point` objects.
Ghost update time for 16M element array on 16 places on Vayu

	time (ms)		
algorithm	**Point**	**inlined**	**speedup**
put	12.3	4.02	3.1
shift	10.5	3.75	2.8

Well-known automatic inlining and scalar replacement transformations[16] will be implemented in future versions of the X10 compiler[17].

3) Byte order in serialization: The current implementation of X10 implements byte-order swapping to ensure message compatibility between places of different architectures. All messages use big-endian byte order. When a place running on a little-endian architecture (e.g. x86) serializes data to be sent to another place, it swaps the byte order to big-endian during serialization. Similarly, when a little-endian place receives a message, it swaps the byte order to little-endian during deserialization.

Swapping byte order adds to the cost of communications, but adds no value for systems where all places share a single byte order (e.g. the *Vayu* cluster of x86 nodes). To avoid this cost, we modified the X10 runtime to avoid byte swapping for homogeneous clusters. If the flag `-HOMOGENEOUS` is set during compilation of the X10 runtime, the byte-order swapping code is replaced with a straightforward memory copy. Table II shows the speedup from avoiding byte swapping on *Vayu*. There is no improvement on *Watson 4P*, as Blue Gene is a big-endian architecture, for which the standard X10 implementation does not perform byte swapping.

TABLE II: Speedup from avoiding byte-order swapping in serialization for a homogeneous cluster.
Microbenchmark timings with shift algorithm as per figure 4 on Vayu

	time (ms)		
number of places	**swap**	**no swap**	**speedup**
2	1.27	1.09	1.17
4	1.91	1.56	1.26
8	3.82	2.96	1.29
16	3.77	3.00	1.26
32	4.21	3.39	1.24
64	4.15	3.42	1.21
128	4.31	3.58	1.20
256	4.36	3.62	1.20
512	4.67	3.97	1.18
1024	4.64	3.95	1.17

The patched version of X10 was used for all benchmarks reported in section IV. We have submitted this enhancement as a patch for inclusion in version 2.3 of the X10 runtime.

A more general solution would perform byte swapping between places only when necessary. For example, places could broadcast architectural information to all other places, and then use this information to determine whether to swap byte order in serialization. Alternatively, each message could include a flag (little/big-endian) which could be used at the receiving place during deserialization.

VII. Conclusion

Active messages allow the straightforward specification of an efficient local-synchronization ghost region update algorithm. We implemented ghost region updates for distributed arrays in the X10 programming language using active messages to combine data transfer and local (pairwise) synchronization. Our results show that using local rather than global synchronization reduces the cost of updating ghost regions on a dynamically threaded architecture. The *shift* algorithm, which trades an increase in synchronization for a reduction in the number of required messages, does not significantly improve performance on either of the architectures we used.

The application and library codes described in this paper are available at http://cs.anu.edu.au/ Josh.Milthorpe/anuchem.html and are free software under the Eclipse Public License.

Acknowledgements

We would like to thank Daniel Frampton and Steve Blackburn from the ANU and David Grove and Olivier Tardieu from

IBM Watson Research Center. This work was partially supported by the Australian Research Council and IBM through Linkage grant LP0989872, and by the NCI National Facility at the ANU.

REFERENCES

[1] MPI Forum, "MPI-2: Extensions to the Message-Passing Interface," University of Tennessee, Knoxville, Tech. Rep., Nov 2003.
[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2005)*, 2005, pp. 519–538, 10.1145/1094811.1094852.
[3] "Chapel language specification," Cray Inc., Tech. Rep. 0.91, April 2012.
[4] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, applications and performance of the Global Arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, p. 203, May 2006, 10.1177/1094342006064503.
[5] B. Palmer and J. Nieplocha, "Efficient algorithms for ghost cell updates on two classes of MPP architectures," in *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, Nov 2002, pp. 197–202.
[6] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.2, 2011.
[7] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification, version 2.2," IBM, Tech. Rep., Jan 2012.
[8] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff, "Constrained types for object-oriented languages," in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA 2008)*, 2008, pp. 457–474, 10.1145/1449764.1449800.
[9] F. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP '10)*, Apr 2010, 10.1145/1953611.1953615.
[10] O. Tardieu, H. B. Lin, and H. Wang, "A work-stealing scheduler for X10's task parallelism with suspension," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP 2012)*, Feb 2012.
[11] J. Milthorpe, V. Ganesh, A. Rendell, and D. Grove, "X10 as a parallel language for scientific computation: practice and experience," in *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)*, May 2011, pp. 1067–1075.
[12] N. Ganesan, B. Bauer, S. Patel, and M. Taufer, "FENZI: GPU-enabled molecular dynamics simulations of large membrane regions based on the CHARMM force field and PME," in *Proceedings of Tenth IEEE International Workshop on High Performance Computational Biology (HiCOMB 2011)*, May 2011.
[13] T. Wen, J. Su, P. Colella, K. Yelick, and N. Keen, "An adaptive mesh refinement benchmark for modern parallel programming languages," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Nov 2007.
[14] B. Chamberlain, S.-E. Choi, T. Hildebrandt, V. Litvinov, G. Titus, J. Lewis, K. Maschhoff, and J. Claridge, "Chapel HPC Challenge entry," Nov 2011, http://chapel.cray.com/presentations/SC11/ ChapelHPCC2011.pdf.
[15] C. Ding and Y. He, "A ghost cell expansion method for reducing communications in solving PDE problems," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001, 10.1145/582034.582084.
[16] S. Fink, K. Knobe, and V. Sarkar, "Unified analysis of array and object references in strongly typed languages," in *Seventh International Static Analysis Symposium (SAS2000)*. Springer, June 2000, pp. 155–174.
[17] D. Grove, February 2012, private communication with X10 design team.