Efficient Update of Ghost Regions Using Active Messages

HiPC 2012
Pune, India
19 December 2012

Josh Milthorpe & Alistair Rendell
Research School of Computer Science
Australian National University
Problem

• Ghost updates for distributed PDE solvers are expensive and a limiting factor in scaling
• Commonly used packages including GA and PETSc use highly synchronised updates

Our contribution

• We used *active messages* to simply express efficient asynchronous updates
We want to go from this...
Overview

• Spatial grids and ghost updates
• The X10 language
• Ghost updates using active messages
• Evaluation
• Conclusion
Overview

• Spatial grids and ghost updates
• The X10 language
• Ghost updates using active messages
• Evaluation
• Conclusion
Grid Applications

- Spatial domain divided into discrete grid
- Target function at each point depends only on neighbouring points
- Iterative
- Used in PDE solvers, comp. fluid dynamics, weather simulation, molecular dynamics...
Grid Applications: Ghost Regions

- Read-only copy of remotely-held data
- Cached to allow computation on boundary elements
- Exchanged many times during computation
Update Algorithms: *Put*

- Exchange overlap regions between each pair of neighbours
- Requires $3^D-1$ messages per place

B. Palmer and J. Nieplocha, "*Efficient algorithms for ghost cell updates on two classes of MPP architectures*," Proc. PDCS 2002
Update Algorithms: *Shift*

- Less communication, more synchronisation
- Send overlaps to neighbours along each axis in turn
**Shift Algorithm**

- Less communication, more synchronisation
- Send overlaps to neighbours along each axis in turn
- Forward ghost data already received
Overview

• Spatial grids and ghost updates
• The X10 language
• Ghost updates using active messages
• Evaluation
• Conclusion
X10 Programming Language - History

- Developed by IBM under DARPA High Productivity Computing Systems program: aim to reduce *time to solution*
- Alongside sister language Chapel (Cray), an asynchronous partitioned global address space (APGAS) language
- ANU linkage project with IBM since 2009:
  - scientific applications in X10
  - high performance managed runtime
X10 Programming Language - Concepts

A type-safe, object-oriented, parallel APGAS language
Basic concepts: places (locality) and asynchronous activities

Place 0

Activities
Local Data

Place (N-1)

Activities
Local Data

Distributed array
Fundamental X10 Constructs

**async**  execute asynchronously at this place

**at(P)**  execute synchronously at place P

**finish**  wait for termination of all spawned activities

**atomic**  execute as if all other activities at this place are suspended

**when(C)**  wait until condition C is true, then execute atomically
Activities and Termination Detection

// start at place0
finish {
    val x = ...;
    async compute(x);
    async at(place1) {
        val y = f(x);
        compute(y);
        async at(place2) {
            val z = g(y);
            ...
        }
    }
}
Activities and Termination Detection

// start at place0

finish {
  val x = ...;
  async compute(x);
  async at(place1) {
    val y = f(x);
    compute(y);
    async at(place2) {
      val z = g(y);
      ...
    }
  }
}

Place 0

finish

compute(x)

Place 1

Place 2
Activities and Termination Detection

// start at place0
finish {
  val x = ...;
  async compute(x);
  async at(place1) {
    val y = f(x);
    compute(y);
    async at(place2) {
      val z = g(y);
      compute(y);
      ...
    }
  }
}
Activities and Termination Detection

```plaintext
// start at place0
finish {
    val x = ...;
    async compute(x);
    async at(place1) {
        val y = f(x);
        compute(y);
        async at(place2) {
            val z = g(y);
            ...
        }
    }
}
```

Diagram:
- Place 0
  - Finish
  - Compute(x)
- Place 1
  - y = f(x)
  - Compute(y)
  - g = f(y)
- Place 2
  - ...

---

Milthorpe and Rendell - Ghost updates using active messages
Activities and Termination Detection

// start at place0

```
finish {
  val x = ...;
  async compute(x);
  async at(place1) {
    val y = f(x);
    compute(y);
    async at(place2) {
      val z = g(y);
      ...
    }
  }
}
```

Place 0

Place 1

Place 2

```
finish
compute(x)
  y = f(x)
compute(y)
  g = f(y)
notify termination
...```

Milthorpe and Rendell - Ghost updates using active messages
Activities and Termination Detection

// start at place0
finish {
  val x = ...;
  async compute(x);
  async at(place1) {
    val y = f(x);
    compute(y);
    async at(place2) {
      val z = g(y);
      ...
    }
  }
}
Synchronisation

```javascript
async {
    compute1a(...);
    atomic result.ready = true;
}

async {
    compute1b(...);
    when(result.ready)
        compute2(result);
}
```

Activity a

Activity b

result ready==false
Synchronisation

```javascript
async {
    compute1a(...);
    atomic result.ready = true;
}

async {
    compute1b(...);
    when(result.ready)
        compute2(result);
}
```
Synchronisation

```javascript
async {
    compute1a(...);
    \texttt{atomic} result.ready = \texttt{true};
}

async {
    compute1b(...);
    \texttt{when(result.ready)}
    compute2(result);
}
```

Activity a

Activity b

\texttt{result ready===false}

test

sleep...
Synchronisation

```javascript
async {
    compute1a(...);
    atomic result.ready = true;
}

async {
    compute1b(...);
    when(result.ready)
        compute2(result);
}
```

Activity a

Activity b

- `compute1a()`
- `compute1b()`
- `test`
- `set`
- `result.ready == false`
- `sleep...`
async {
    compute1a(...);
    atomic result.ready = true;
}

async {
    compute1b(...);
    when(result.ready)
        compute2(result);
}
Arrays, Regions and Distributions

```scala
val r:Region(2) = 0..15 * 0..3;
val a = new Array[Double](r);

for([i,j] in r) async {
    a(i,j) = ...;
}

val d:Dist(2) = Dist.makeBlock(r);
val da = DistArray.make[Double](d);
```
ANUChem: Chemistry Applications in X10

- Developed a suite of scientific applications to test expressiveness, performance and scalability of X10

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartree-Fock (HF)</td>
<td>6000</td>
</tr>
<tr>
<td>Particle-Mesh Ewald (PME)</td>
<td>1000</td>
</tr>
<tr>
<td>Fast Multipole Method (FMM)</td>
<td>3500</td>
</tr>
<tr>
<td>FT-ICR simulation</td>
<td>900</td>
</tr>
<tr>
<td>shared code</td>
<td>2500</td>
</tr>
</tbody>
</table>

- Experiences shared with language designers

Overview

• Spatial grids and ghost updates
• The X10 language
• Ghost updates using active messages
• Evaluation
• Conclusion
Implementing Ghost Updates

- New classes in `x10.array` and new API on `DistArray`:
  - `sendGhostsLocal()` - single place
    send ghost data to neighbouring places
  - `waitForGhostsLocal()` - single place
    wait for ghost data to be received from neighbouring places
  - `updateGhosts()` - global operation
    update ghost regions for the entire array
Synchronisation Requirements

Ghost update in iterative computation:

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

Must ensure ghosts are fully received before computation begins:

- Use implicit synchronisation through send/recv (PETSc)
- Surround with collective barrier (Global Arrays)
Active Messages for Local Synchronisation

Active messages combine data transfer and synchronisation
Phase counter per array at each place – may not advance >1 phase

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

Conditional atomic block waits until all ghosts have been received:

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

Allows *split-phase* updates
Overview

• Spatial grids and ghost updates
• The X10 language
• Ghost updates using active messages
• Evaluation
• Conclusion
Performance Evaluation

• Evaluated put vs. shift algorithm; local vs. global sync; applications (lattice Boltzmann & PME)
• Vayu: Sun X6275 Nehalem / QDR IB
• Watson 4P: Blue Gene/P
• Native X10 2.2.2.1, one thread per place
• Global Arrays v5.1
ghost update weak scaling for number of places

1M doubles per place

mean time per update (ms)

number of places

Watson 4P - shift
Watson 4P - put
Vayu - shift
Vayu - put
ghost update weak scaling on Vayu

mean time per update (ms)

number of places

0
1
2
3
4
5
6
1
2
4
16
64
256
1024

global barrier
local synchronization
Lattice Boltzmann on Nehalem + QDR IB cluster
1024*1024 grid, Reynolds no. \approx 850

- X10 – total
- X10 – update ghosts
- GA – total
- GA – update ghosts

Global arrays crashed above 64 places
Language Implementation Issues

• Atomic blocks: implemented using place-wide lock
  => Use fine-grained locks or software transactional memory

• Inlining of Point objects:
  arrays of dynamic rank generate high overhead
  => Automatic inlining and scalar replacement transformations

• Byte-order swapping in serialisation:
  swap to network byte order, unnecessary for uniform clusters
  => \(-\text{HOMOGENOUS}\) flag added to X10 runtime build (X10 2.3)
  => use architectural info to swap only when necessary
  => include endian flag with each message
Conclusions and Future Work

Unnecessary synchronisation in ghost region updates limits scaling for distributed applications that use spatial grids.

We implemented ghost region updates using *active messages* to combine data transfer and local synchronisation.

Local synchronisation reduces the cost on a dynamically threaded architecture.

Update algorithm (put vs. shift) not important.

Conclusions and Future Work

Unnecessary synchronisation in ghost region updates limits scaling for distributed applications that use spatial grids.

We implemented ghost region updates using *active messages* to combine data transfer and local synchronisation.

Local synchronisation reduces the cost on a dynamically threaded architecture.

Update algorithm (put vs. shift) not important.


Questions?
Particle Mesh Ewald on Blue Gene/P and Nehalem + QDR IB
80° water box, 51396 atoms, grid size=64