Porting MPI based HPC Applications to X10

Hiroki Murata, Michihiro Horie, Koichi Shirahata†, Jun Doi, Hideki Tai, Mikio Takeuchi, Kiyokuni Kawachiya

IBM Research – Tokyo, †Tokyo Institute of Technology

This material is based upon work supported by the Department of Energy under award DE-FOA-0000619.
Motivation: feasibility study X10 as the language for HPC

- There are many HPC applications that are written in message passing style.
- Re-implementing those applications with native X10 communication primitives may outperform the original code, but it takes time to implement.
- Porting those applications to X10 while preserving the original code structure is possible, but the performance may degrade.

- Are there any X10 idioms and modification patterns to avoid the performance degradation?
Contributions

- Implementation of two proxy applications in X10 tuned to achieve performance nearly comparable to the original code
- X10 idioms and modification patterns which include APGAS code that can replace MPI-based communication patterns
- Performance evaluations of the code
CoMD (Co-design for Molecular Dynamics)

- An application of Molecular Dynamics (MD) simulation
- Simulation of materials with inter-atomic potential
  - Supported potentials
    - Lennard-Jones potential
    - Embedded-Atom Method potential
  - Forces between all atom pairs within the cutoff distance are evaluated.
- Cartesian spatial decomposition of atoms across nodes
- Atoms on a node are assigned to cubic link cells which are slightly larger than the cutoff distance
- Implementation
  - C+MPI
MCCK (Monte Carlo Communication Kernel)

- An application of a domain decomposed particle tracking algorithm
- Simulation of randomly-moving particles
  - Initially a number of particles exist in each physical node
  - A particle in a node can move into one of the neighbor nodes
  - As time advances, particle fission occurs
- Implementation
  - C+MPI
Porting strategy

- Keep the original code structure as much as possible
- Map an MPI process to an X10 place
- Map a C array to an X10 Rail
- Map a C struct to an X10 class
- Transform MPI point to point communication to Rail.asyncCopy method call
- And apply modifications to achieve comparable performance
Basic modifications

- Memory allocation
- Object access
- Others
Basic modifications (Memory allocation)

1. Replace local variable with field variable
2. Move allocation statements in loop to out of loop
3. Allocate object on stack (with @StackAllocate, experimental)
**Basic modifications (Object access)**

1. Use val as much as possible
2. Unify common expressions
3. Hoist loop invariant
4. Flatten a multi-dimensional array index and its loop
5. Replace short array with variables

```c
// C code
int force(SF* s) {
    ...
    for (int iBox=0; ...) {
        ...
        for (int iOff= ...) {
            ...
            s->atom->p[iOff][0] -= dP * dr / r;
            s->atom->p[iOff][1] += dP * dr / r;
            ...
        }
    }
}
```

```x10
// X10 code
public def force(s:Types.SF):Int {
    val pT = new Cell[Double](0.0);
    ...
    for (var iBox:Int=0n; ...) {
        ...
        for (var iOff:Int=...) {
            ...
            val tmp2 = dP.value * dr / r;
            s.atom.p(iOff)(0) -= tmp2;
            s.atom.p(iOff)(1) += tmp2;
            ...
        }
    }
}
```

1. Use val as much as possible
2. Unify common expressions
3. Hoist loop invariant
4. Flatten a multi-dimensional array index and its loop
5. Replace short array with variables
Basic modifications (Others)

1. Set class as final
2. Forcibly inline method called in loop (with @Inline)
3. Prepare dedicated method

// C code
typedef struct {
    double x; double y; double z;
    int absorbed;
} Particle;

int compare(const void *pp1, const void *pp2) {
    Particle *p1 = (Particle *) pp1;
    Particle *p2 = (Particle *) pp2;
    if (p1->absorbed && !p2->absorbed) return 1;
    else if (!p1->absorbed && p2->absorbed) return -1;
    else return 0;
}

static int pack() {
    Particle *p = mc->particles; int np = mc->nparticles;
    int count;
    qsort(p, np, sizeof(Particle), compare);
    for (count = 0; count < np; ++count)
        if (p[count].absorbed) break;
    return count;
}

// X10 code
public class Particle {
    var x:Double; var y:Double; var z:Double;
    var absorbed:Boolean;
    static def compare(p1:Particle, p2:Particle):Int { 
        if (p1.absorbed && !p2.absorbed) return 1n;
        else if (!p1.absorbed && p2.absorbed) return -1n;
        else return 0n;
    }
}

public class MC_Cycle {
    private def pack():Int {
        val p = mc.particles(); val np = mc.nparticles as Long;
        var count:Int;
        //RailUtils.qsort[Particle](p,0,np-1,(x:Particle,y:Particle)=>Particle.compare(x,y));
        myqsort(p, 0, np-1);
        for (count = 0n; count < np; ++count)
            if (p(count).absorbed) break;
        return count;
    }

    static def myqsort(a:Rail[Particle], lo:Long, hi:Long):void {
        if (hi <= lo) return;
        var l:Long = lo - 1; var h:Long = hi;
        var temp:Particle;
        while (true) {
            while (Particle.compare(a(++l), a(hi))<0) ;
            while (Particle.compare(a(hi), a(--h))<0 && h>lo) ;
            if (l >= h) break;
            temp = a(l); a(l) = a(h); a(h) = temp;
        }
        temp = a(l); a(l) = a(hi); a(hi) = temp;
        myqsort(a, lo, h-1n); myqsort(a, l+1n, hi);
    }
}
Modifications related to MPI

- **Point-to-Point communication**
  - CoMD uses the `send-receive` operation.
  - MCCK uses the sequence of `MPI_BARRIER`, `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall`. This sequence can be implemented by executing barrier synchronization and then moving to the other place and accessing the data.

- **Collective operations**
  - Almost all of the collective operations are prepared in the X10 "Team" API.
  - However, `MPI_MINLOC` and `MPI_MAXLOC` are not.
Modifications related to MPI (P2P send-receive)

// C code
void sendReceive(void* sendBuf, int sendLen, int dest, 
void* recvBuf, int recvLen, int source, 
int* recvLen) {
MPI_Status status;
MPI_Sendrecv(sendBuf, sendLen, MPI_BYTE, dest, 0, 
recvBuf, recvLen, MPI_BYTE, source, 0, 
MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_BYTE, recvLen);
}

// X10 code
public class Communicator {
val startLatch = PlaceLocalHandle[Rail[MyLatch]]; 
val finishLatch = PlaceLocalHandle[Rail[MyLatch]]; 

def this() {
    this.startLatch = 
        PlaceLocalHandle.make[Rail[MyLatch]](Place.places(), 
() => new Rail[MyLatch](Place.MAX_PLACES));
    this.finishLatch = 
        PlaceLocalHandle.make[Rail[MyLatch]](Place.places(), 
() => new Rail[MyLatch](Place.MAX_PLACES));
}

public def sendReceive[T](
    sendBuf: PlaceLocalHandle[Rail[T]], 
    sendLen: PlaceLocalHandle[Rail[Int]], dest:Int, 
    recvBuf: PlaceLocalHandle[Rail[Int]], recvLen: Int, 
    source:Int, recvLen: PlaceLocalHandle[Rail[Int]]) {
val me = here.id() as Int;
if (me == dest && me == source) {
    val len = sendLen()(0) as Long;
    // Notify the "dest" that "me" is ready to send
    startLatch(me).release();
    // 1. Trigger asyncCopy(me -> dest)
    at (Place.place(dest)) async {
        // Notify the "dest" that "me" is ready to send
        startLatch(me).release();
    }
    // 2.1. Wait for the "source" ready to send
    val recvBufRef = GlobalRail(recvBuf());
    val recvdLenRef = GlobalRail(recvLen());
    // Wait for a notification from the "source"
    startLatch(source).await();
    // Now both recvBuf at "me" and sendBuf
    // at the "source" are ready for asyncCopy
    // 2.2. Perform asyncCopy(source -> me)
    at (Place.place(source)) async {
        val len2 = sendLen()(0) as Long;
        finish {
            Rail.asyncCopy[T](sendBuf(), 0, recvBufRef, 0, len2);
            Rail.asyncCopy[Int](sendLen(), 0, recvdLenRef, 0, 1);
            // Notify the "source" the completion
            // of asyncCopy(source -> me)
            finishLatch(me).release();
        }
    }
    // 3. Wait for a notification of the completion
    finishLatch(dest).await();
} else {
    // Notify the "dest" that "me" is ready to send
    finish {
        // 1. Trigger asyncCopy(me -> dest)
        at (Place.place(dest)) async {
            // Notify the "dest" that "me" is ready to send
            startLatch(me).release();
        }
    }
}}
Modifications related to MPI (Collective MPI_MINLOC)

// C code
typedef struct {
    double value;
    int rank;
} RankReduceData;

void minRankDouble(
    RankReduceData* sBuf, RankReduceData* rBuf, int count) {
    MPI_Allreduce(sBuf, rBuf, count,
                  MPI_DOUBLE_INT, MPI_MINLOC, MPI_COMM_WORLD);
}

// X10 code
public class Communicator {
    static class RankReduceData {
        var value:Double = 0.0;
        var rank:Int = 0n;
    }

    public def minRankDouble(sendBuf:Rail[RankReduceData],
                              recvBuf:Rail[RankReduceData], count:Int):void {
        val sendBuf2 = new Rail[Double](count);
        val recvBuf2 = new Rail[Double](count);
        for (var i:Int = 0n; i < count; i++) {
            sendBuf2(i) = sendBuf(i).value;
        }

        team.allreduce[Double](sendBuf2, 0, recvBuf2, 0,
                              count as Long, Team.MIN);
        for (var i:Int = 0n; i < count; i++) {
            recvBuf(i).rank = team.indexOfMin(sendBuf(i).value,
                                              sendBuf(i).rank);
            recvBuf(i).value = recvBuf2(i);
        }
    }
}
Evaluation (CoMD w/ Lennard-Jones potential)

Scales well up to 392 nodes. Up to 11% slower than the original.

The performance loss is due to communication factor.

The data and the received length are sent with two async copies, rather than with a single send-receive in the original.

Weak scaling performance of CoMD with Lennard-Jones (LJ) potential
(Problem size: 256000 atoms/node)

IBM Power775 (13x POWER7 3.84 GHz (8x4 HWT, 128GB mem)), RHEL 6.2
X10 2.4.1 native backend (compile option: -x10rt pami -O -NO_CHECKS)
C/C++ compiler: XL C V12.1 (compile option: -O3 -qinline)
Communication runtime: PAMI (X10) and MPI (Original)
Evaluation (CoMD w/ Embedded-Atom potential)

Scales well up to 392 nodes. Up to 25% slower than the original.

The performance loss is due to both calculation and communication factors.

For the calculation, a table (an array) is looked up, which is slower than in the original.

For the communication, the data and the received length are sent with two async copies, rather than with a single send-receive in the original. Furthermore, it communicates more data than LJ potential version.
Evaluation (MCCK)

Scales well up to 360 places (limited by memory size). Up to 30% faster than the original for one place. Up to 17% and on average 1.8% slower than the original on multiple places.

The performance gain on one place is due to the fast calculation and the slow communication.

The calculation (i.e. sorting) is fast, since it is implemented as a dedicated method and the comparator function is inlined. Furthermore it sorts an array of reference rather than an array of struct.

The communication is slow, since it emulates point-to-point communications. The communication time varies irregularly, its rate of variation is similar to that of the original.
Conclusion

- Explained X10 idioms and modification patterns for porting two proxy applications, CoMD and MCCK, onto X10
- Those performance was measured on an IBM Power 775 cluster
  - X10 port of those applications scaled comparably with the original.
  - X10 ports of CoMD with Lennard-Jones and Embedded-Atom Method potentials were 11% and 25% slower than the original, respectively.
  - X10 port of MCCK was close to the original.