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

[†]{mrthrk,horie,doichan,mtake,kawatiya}@jp.ibm.com [‡]koichi-s@matsulab.is.titech.ac.jp

Abstract

X10 is a high-productivity programming language that internally supports parallel and distributed computing. X10 is based on an APGAS (Asynchronous Partitioned Global Address Space) programming model. Applications written in X10 can run on multiple *place*s, which are abstractions of computation nodes, create *activities* to perform parallel computations in the same place by using `async` statements, or perform distributed computing by changing the execution places by using `at` statements. In this paper, we report on our experiences in porting typical applications for high-performance computing to X10. These applications were originally written in C with MPI, and the ported applications were written in pure X10. We confirmed that the X10 port of these applications showed comparable performance and scalability in a large-scale, parallel, and distributed environment such as Power® 775, which is one of IBM®'s latest supercomputers. We also report several techniques to obtain good performance in X10 for typical coding patterns such as array accesses, broadcasts, and data exchanges of ghost regions of data.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming

General Terms Languages, Performance

Keywords X10, APGAS, SPMD, MPI, HPC, high performance computing, proxy applications, CoMD, MCCK

1. Introduction

For simulating real parallel and distributed applications such as analyzing nuclear reactors, community groups called *co-design centers* provide parallel and distributed applications, which are called *proxy application*s (or *proxy app*s). Proxy apps can be used to observe and explore parallelism and memory layouts in distributed environments.

X10 is a modern object-oriented programming language that introduces new constructs significantly simplify scale-out programming based on an APGAS (Asynchronous Partitioned Global Address Space) programming model. One fundamental goal of X10 is to enable scalable, high-performance, high-productivity programming for high-end computers for traditional numerical computation workloads such as weather simulation, molecular dynamics, and particle transport problems.

We are studying the applicability of X10 by using proxy apps. These proxy apps were developed by the co-design centers, named the Center for Exascale Simulation of Advanced Reactors (CESAR) [1], the Exascale Co-design Center for Materials in Extreme Environments (ExMatEx) [2], and the Center for Exascale Simulation of Combustion in Turbulence (ExaCT) [3]. Serial versions of these proxy apps were implemented in C, C++, Fortran, Python, etc. To parallelize the proxy apps, MPI [4], OpenMP [5], or OpenCL [6] was used in many cases.

In this paper, we show our experience to port two applications to X10. One of the proxy apps is CoMD [7], which is an application of classical molecular dynamics (MD) algorithms used in material science. The other is MCCK [9], which is an application of Monte Carlo simulation for investigating the communication cost of the domain decomposed particle tracking algorithm. We then compared the original apps and the X10 ports of the apps, and confirmed that the X10 ports of these applications showed comparable performance and scalability in a large-scale, parallel and distributed environment such as Power® 775, which is one of IBM®'s latest supercomputers. We also report several techniques to obtain good performance in X10 for typical coding patterns such as array accesses, broadcasts, and data exchanges of ghost regions of data.

Here are the main contributions of our work:

- CoMD and MCCK proxy applications in X10 tuned to achieve performance nearly comparable to the original code.
- Porting tips which include APGAS code that can replace MPI-based communication patterns.
- Performance evaluations of the code on an IBM® Power® 775 cluster.

2. Background

2.1 X10 Language

X10 features flexible support for concurrency, distribution, and locality. X10 uses a model called APGAS (Figure 1). In this model, a global address space is divided into small regions called *place*s. X10 introduced places as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 computation can run across a large collection of places. Each place hosts some of the data and runs one or more *activities*. Activities are extremely lightweight threads of execution. An activity may synchronously (and atomically) use one or more memory locations in its place and create other activities with the `async` statement, exploiting the performance of current symmetric multiprocessor (SMP) technology. It can move to other places by using the `at` statement to access data in those places. `PlaceLocalHandle`, an X10 type is used to access data in other places. It bundles multiple remote references to the objects at different places.

2.2 Applications

We ported two applications to X10 and compared each of them to the original versions.

* Hideki Tai is currently self-employed. He can be reached at hidekit55@gmail.com.

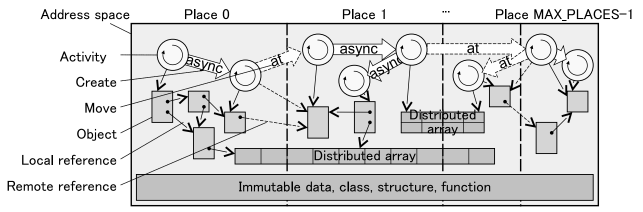

Figure 1. APGAS model

2.2.1 CoMD

CoMD (Co-design for Molecular Dynamics) is an application for Molecular Dynamics (MD) workloads as used in materials science. MD is a well-known simulation for the physical movements of atoms and molecules. MD is widely used in the areas of chemical physics, material science, biology, and engineering.

As described in [7], the methods of MD simulation involve the evaluation of the force acting on each atom due to all of the other atoms in the system and the numerical integration of the Newtonian equations of motion for each of the atoms.

The reference version of CoMD 1.1 written in C with MPI and a geometric domain decomposition SPMD programming model. CoMD uses a linked-cell structure to determine the interaction neighborhood and supports both the Lennard-Jones potential and the Embedded-Atom Method potential.

We used the CoMD source code revision 90def13d8c at [8].

2.2.2 MCCK

The Monte Carlo Communication Kernel (MCCK) was developed to simulate particle movement in the domain of a nuclear reactor. In MCCK, initially a number of particles exist in each physical node. A particle at a node can move to another node that is next to its current node. In addition, as time advances, nuclear fission occurs, so the number of particles to calculate is gradually reduced.

A main use of MCCK is to investigate the communication costs when data is exchanged among physical nodes. The original MCCK was implemented in C with MPI.

We used the MCCK source code at [9].

2.2.3 Basic Porting Strategy

For porting these applications, we used the following strategy and kept the original code structure as much as possible: mapping an MPI process to an X10 place; transforming C array to X10 `Rail` which is an abstraction of fixed-sized indexed storage; mapping C struct to X10 class (X10 struct cannot be used because it is immutable); and transforming MPI point to point communication to `Rail.asyncCopy` method call. However, because it was not possible to achieve comparable performance, we applied some optimizations. In the next two sections, we explain our optimizations and porting tips: basic modifications in Section 3 and modifications related to MPI in Section 4.

3. Basic Modifications

This section explains the basic modification patterns to port the applications described in Section 2.2 to X10. Figures 2 and 3 show the examples of the basic modification patterns. Figure 2 shows the original C code and Figure 3 shows the X10 code.

3.1 Memory Allocation

C/C++ programs can use complex data structure as local variables on stack. This does not degrade the application performance, even in loops. However, allocating complex data structures on the heap in loops may degrade the performance on object oriented languages.

Replace local variable with field variable Memory allocation in a method called in loop may degrade the application performance, especially when large objects are allocated. In this case, replacing a local variable with a field variable will improve the performance. A transformation example for Line 3 of Figure 2 becomes Line 2 of Figure 3.

Move allocation statements in loop to out of loop Memory allocation in a loop also degrades the application performance. In this case, moving the allocation statements in the loop outside of the loop will improve the performance. A transformation example is Line 25 of Figure 2 becomes Line 10 of Figure 3.

Allocate object on stack (experimental) Adding `@StackAllocate` annotation to allocation sites in a loop may improve the application performance by reducing the overhead of allocation. Although it is experimental, it would be an alternative when the above rewritings are hard to apply.

3.2 Object Access

Use val as much as possible In X10, there are two type of variable declarations, `val` and `var`. When `val` is used, X10 compiler analyzes its type and generates better code. Lines 2, 5, 10 and others of Figure 3 are examples.

Unify common expressions When porting C/C++ applications onto X10, a `Cell` is sometimes used to replace a pointer parameter. Accessing the content in a `Cell` is slower than accessing the variable in the original C code. Unifying common expressions including `Cell` access avoids this performance degradation. A transformation example involves Lines 13–14 of Figure 2 becoming Lines 16–18 of Figure 3. This is similar for `PlaceLocalHandle`.

Hoist loop invariant Since C struct is mapped to X10 class, nested C structs are transformed to nested X10 classes. The generated code from the nested X10 classes is currently not optimized by the backend compiler. Hoisting them in a loop can improve the application performance. A transformation example involves transforming *s->atom->r* on the right side of Line 21 of Figure 2 to *r* on the right sides of Lines 26, 28, and 30 with Line 20 of Figure 3.

Flatten a multi-dimensional array index and its loop In these ports, C array is transformed to `Rail`. The generated code for a nested `Rail` in X10 currently does not fit the pattern for optimization by the backend compiler. Flattening a multi-dimensional array index can improve the application performance. An example involves transforming *r[jOff][m]* on the right side of Line 21 of Figure 2 into *r(...)* on the right side of Lines 26, 28, and 30 of Figure 3.

Replace short array with variables Replacing a short array with variables can improve the performance by eliminating the array allocation and array index calculations. An example involves transforming *d[m]* on the left side of Line 21 of Figure 2 to *d0*, *d1*, *d2* on the left side of Lines 26, 28, and 30 of Figure 3.

The X10 compiler depends for the code optimization on the backend compiler. However the generated code for the latter four modifications in this section by X10 currently do not fit the pattern for optimization by the backend compiler, but it is possible to optimize them by using X10. We plan to extend X10 to support these optimizations.

3.3 Others

Set class as final Setting a class as final may improve the application performance by inlining the class method.

```
void sort(Link* boxes, ...) {
    int nA = boxes->nA[iBox];
    Msg tmp[nA];
    ...
}

int force(SF* s) {
    ...
    for (int iBox=0; ...) {
        ...
        for (int iOff= ...) {
            ...
            s->atom->p[iOff][0] -= dP * dr / r; // dP is double
            s->atom->p[iOff][1] += dP * dr / r;
            ...
            for (int jOff= ...) {
                double d[3];
                ...
                double r2 = 0.0;
                for (int m=0; m<3; m++) {
                    d[m] = s->atom->r[jOff][m];
                    r2 += d[m] * d[m];
                }
                ...
                double pT;
                ip(&pT);
                for (int m=0; m<3; m++) {
                    s->atom->f[jOff][m] += pT * d[m];
                }
            }
        }
    }
}

void ip(double* df) {
    *df = ...
}
```

Figure 2. Original C code for memory allocation and object access

Forcibly inline method called in loop Adding `@Inline` annotation to any methods called in a loop may improve the application performance by inlining the methods. For example, this is suitable to transform macros with parameters.

Prepare dedicated method X10 sometimes prepares very generalized APIs, but these have some overhead. For critical operations, a dedicated method is better for performance.

4. Modifications related to MPI

This section describes modifications related to MPI.

4.1 Initialization

MPI applications typically use the MPI functions through locally defined functions, for example, initCommunicator() for MPI_Init(), destroyCommunicator() for MPI_Finalize(), and so on. The main() starts by calling initCommunicator() and ends by calling destroyCommunicator(), and any initializations are done between initCommunicator() and destroyCommunicator().

X10 provides `broadcastFlat()` API to execute its parameter in SPMD style. However, data used independently by each place should be created as `PlaceLocalHandle` before parallel sections start. The main() function can be transformed as shown in Figures 4 and 5. First, the constructor of *Buffers* allocates buffers at all places. Then, the original sequence between *initCommunicator()* and *destroyCommunicator()* is passed as a function to `broadcastFlat()` via *initCommunicator()*.

4.2 Point-to-Point Communication

The sending and receiving of messages by processes is the basic MPI communication mechanism. MPI provides blocking send and receive operations, nonblocking communications, and send-receive operations. MPI blocking operations are two-sided which synchronize the sender and the receiver, but `at` statement in X10 is one-sided which does not synchronize activities in source and destination places. MPI nonblocking operations have accompanying functions to query the status of the operations, but `async` statement in X10 does not have a query mechanism. Therefore, rewriting MPI point-to-point communications on X10 is not a simple task.

```
public class Calc {
    val tmp = new Rail[Msg](Links.MAXA);

    public def sort(boxes:Links.Link, ...):void {
        val nA = boxes.nA(iBox);
        ...
    }

    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; // dP:Cell[Double]
                s.atom.p(iOff)(0) -= tmp2;
                s.atom.p(iOff)(1) += tmp2;
                ...
                val r = s.atom.r;
                val f = s.atom.f;
                for (var jOff:Int = ...) {
                    val jOff3 = jOff*3;
                    ...
                    var r2:Double = 0.0;
                    val d0 = r(jOff3);
                    r2 += d0 * d0;
                    val d1 = r(jOff3+1);
                    r2 += d1 * d1;
                    val d2 = r(jOff3+2);
                    r2 += d2 * d2;
                    ...
                    ip(pT);
                    val tmp3 = pT.value;
                    f(jOff3) += tmp3 * d0;
                    f(jOff3+1) += tmp3 * d1;
                    f(jOff3+2) += tmp3 * d2;
                }
            }
        }
    }

    @Inline public static def ip(df:Cell[Double]) {
        df.value = ...
    }
}
```

Figure 3. Memory allocation and object access

CoMD uses the send-receive operation for its communications. The send-receive operation sends a message to one process and receives another message from another process. This is very useful for executing a shift operation across a chain of processes. However, to correctly order the sends and receives is important to prevent cyclic dependencies that may lead to deadlock.

Figure 6 is a coding example of send-receive operation using *MPI_Sendrecv()* and *MPI_Get_count()*. Its parameters, *sendBuf*, *sendLen*, *dest*, *recvBuf*, *recvLen*, and *source* specify the address of the send buffer, the length of send buffer, the process id of the destination process, the address of the receive buffer, the length of the receive buffer, and the process id of the source process, respectively. The recvdLen returns the length of the received data. *MPI_Sendrecv()* is the main part and its parameters specify the send and receive buffers, the ranks of the destination and source processes, and so on. *MPI_Get_count()* is used to obtain the length of the received message.

X10 doesn't provide an API compatible with *MPI_Sendrecv()*, but provides APIs to copy data remotely and APIs to synchronize activities separately. Therefore a programmer can implement *MPI_Sendrecv()* with these API calls. Figure 7 is a coding example of send-receive operation in X10. Its parameters' names and meanings are the same as those of C version. *MyLatch* is our own implementation of latch to release multiple activities under waits. First, it notifies the destination that it is ready to send data

```
int main(int argc, char** argv) {
    initCommunicator(&argc, &argv);
    // initialization part
    // main loop
    // finalization part
    destroyCommunicator();
    return 0;
}

void initCommunicator(int* argc, char*** argv) {
    MPI_Init(argc, argv);
}

void destroyCommunicator() {
    MPI_Finalize();
}
```

Figure 4. Original C code for initialization

```
public class Application {

    public static def main(args:Rail[String]):void {
        val comm = new Communicator(args);
        initCommunicator(args, comm,
            (args:Rail[String], comm:Communicator)=>{
            // initialization part
            // main loop
            // finalization part
        });
        destroyCommunicator();
    }

    static def initCommunicator(args:Rail[String],
        comm:Communicator,
        body:(Rail[String], Communicator)=>void):void {
        PlaceGroup.WORLD.broadcastFlat(()=>{body(args, comm);});
    }

    static def destroyCommunicator():void {
    }
}

public class Communicator {
    // Class for preparing PlaceLocalHandle for buffers
    // and communication

    public this(args:Rail[String]) {
        // execute a part of the original initialization
        // to calculate the number and size of buffers

        // then allocate and initialize PlaceLocalHandle
        // for the buffers
    }
}
```

Figure 5. Initialization

(Lines 31–33). Then it waits until the source becomes ready to send the data (Line 40), and its receives the data from the source and notifies the source that the data reception is complete (Lines 45–56). `Rail.asyncCopy()` in these lines is an API to copy data between a local `Rail` to a `GlobalRail` (a rail at remote place) asynchronously. `Rail.asyncCopy()` in Lines 48–49 copies data of sendBuf at source place to local recvBuf, and one in Lines 50–51 copies sendLen at source place to local recvdLen. Then it waits for the notification of data reception completion from the destination (Line 60). We are considering to provide this send-receive implementation as a library.

MCCK uses the sequence of MPI_Barrier, MPI_Isend, MPI_Irecv, and MPI_Waitall for its communication. This sequence can be implemented by executing barrier synchronization and then moving to the other place by using the `at` statement asynchronously with the `async` statement and accessing the data in the new place.

4.3 Collective Operations

Almost all of the collective operations are already prepared in the X10 "Team" API [10], but the APIs corresponds to two predefined operators, *MPI_MINLOC* and *MPI_MAXLOC*, of global reduce operations are not defined. The operator *MPI_MINLOC* is used to compute a global minimum and also an index attached to the minimum value. *MPI_MAXLOC* similarly computes a global maximum and index. Instead of these operators, the X10 "Team" API offers `Team.indexOfMin()` and `Team.indexOfMax()` that return the index of the place with the minimum or maximum value, respectively. The global reduce operations with *MPI_MINLOC* or *MPI_MAXLOC* can be implemented by combining them with global reduce operations of the X10 "Team" API, `Team.reduce()` and `Team.allreduce()`. Figure 8 and 9 is the transformation example from *MPI_Allreduce* with *MPI_MINLOC* in C to `Team.allreduce()` with `Team.indexOfMin()` in X10.

5. Evaluation

This section evaluates the X10 ports of the proxy apps compared to the original implementations. The experimental environment was an IBM® Power®775 server. It has 13 nodes and each node has 32 cores of POWER7® at 3.84 GHz with 128 GB memory. The OS was Red Hat Enterprise Linux® Server release 6.2. The X10 2.4.1 and the IBM® XL C V12.1 compilers were used. X10 was used in a native back-end. The compile options for the original code were "`xlC_r -O3 -qinline`", and for the native X10 we used "`x10c++ -x10rt pami -O -NO_CHECKS`". The source code of CoMD ported to X10 is available at [11], and MCCK is at [12].

The top and middle graphs in Figure 10 show the weak scaling performance of CoMD. The problem size is 256,000 atoms/node. CoMD implemented in X10 with "Embedded-Atom Method potential" is from 9% to 25% slower than the original version (the top graph of Figure 10), and with "Lennard-Jones potential" it is from 6% faster to 11% slower than the original (the middle graph of Figure 10).

The performance loss of the "Embedded-Atom Method potential" version in X10 is due to both calculation and communication factors. For the calculations, the "Embedded-Atom Method potential" version uses a table to calculate the potential of the atoms. The table values are stored in arrays and the potential calculation accesses the arrays and this is slower time than the original. For the communications, the CoMD X10 implementation uses the send-receive operation shown in Figure 7. This send-receive operation does two remote copies to send the data (Lines 48–49) and to send the received length (Lines 50–51), while the original C implementation needs only one remote copy. We are considering adding another API to do these two copies in one step.

The performance reduction of the "Lennard-Jones potential" version in X10 is due to the communications. Since it uses less communications than the "Embedded-Atom Method potential" version, its performance degradation due to communication overhead is smaller.

However, both potential versions scale well up to 392 (7x7x8) places. Since each of the MPI processes is sequential, we can run up to 416 (13x32) places.

The bottom graph of Figure 10 shows the relatively weak scaling of MCCK. The problem size is 20,000,000 particles/node with 0.2 leakage. MCCK implemented on X10 is from 30% faster to 17% slower than the original version, and on average 1.8% slower.This performance is due to the calculation. Since MCCK is a benchmark for communication, its calculation part decreases the particles stochastically, packs the remaining particles in an array while sorting, then unpacks the exchanged particles in the array while sorting. Based on the basic modifications explained in Section 3, the sort is implemented as a dedicated method, and the compare function used by the sort is inlined. While the original version sorts an array of structs, the X10 version sorts an array of pointers (to objects). In the results, the calculation part is accelerated. The communications are slower than in the original version,

```
void sendReceive(void* sendBuf, int sendLen, int dest,
                 void* recvBuf, int recvLen, int source,
                 int* recvdLen) {
   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, recvdLen);
}
```

Figure 6. Implementation example of send-receive operation in C

since it emulates point-to-point communication while moving to the other place and accessing its data. However the communication time varies irregularly, its rate of variation is similar to that of the original version. It scales well up to 360 places,but is limited by memory capacity.

6. Related Work

Karlin et al [13] ported the proxy application LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [14] to four emerging programming models, Chapel [15], Charm++ [16], Liszt [17], and Loci [18], and comparing the performance with the original implementations, OpenMP and MPI. Chapel achieves more than 80% efficiency at 16 cores although its single-core performance is significantly worse compared to OpenMP. Charm++ weak scales extremely well and its performance is comparable to the MPI implementation. The Liszt MPI back end's on-node performance was about 50% worse than the native MPI implementation. Inserting native C++ code into their Liszt program, the performance improved to within 10% of the MPI implementation. Loci's strong scaling on a node outperforms OpenMP by up to 15%. Overall, its scaling is a bit worse than OpenMP. Weak scaling performance of the Loci implementation comes close to that of MPI implementation. In our results, the performance of CoMD ported onto X10 is 11% and 25% worse than that of the MPI implementation and MCCK ported to X10 is comparable to the MPI implementation, are comparable to their results.

Since PGAS programs may have many fine-grained shared accesses that lead to performance degradation, much research has been done towards improving the performance. For example, data coalescing [19–21, 24], splitting request and completion of shared accesses [21–23], and so on. Rewriting applications from scratch for the X10 APGAS model and applying those optimizations may result in higher than original performance. However, we didn't rewrite any applications from scratch nor use any of those optimizations. We ported the applications directly in this study.

7. Conclusion

We described the rewriting patterns for porting two proxy applications, CoMD and MCCK onto X10. The performance was measured on an IBM® Power® 775 cluster. The performance of CoMD ported to X10 with the Embedded-Atom Method potential was from 9% to 25% slower, and CoMD in X10 with the Lennard-Jones potential was from 6% faster to 11% slower, compared to the original. The performance of MCCK ported to X10 is quite close to the original. Those applications ported to X10 scaled well and was comparable performance with the original. All of the ported code are available at X10 sites. We are planning to provide the utility libraries of common porting patterns.

```
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[T]], recvLen:Int,
        source:Int, recvdLen:PlaceLocalHandle[Rail[Int]]) {
        val me = here.id() as Int;
        if (me == dest && me == source) {
            val len = sendLen()(0);
            Rail.copy[T](sendBuf(), 0, recvBuf(), 0,
                len as Long);
            recvdLen()(0) = len;
        } else {
            finish {
                // 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(recvdLen());
                // 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);
                    finish {
                        Rail.asyncCopy[T](sendBuf(), 0,
                            recvBufRef, 0, len2 as Long);
                        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
                // of asyncCopy(me -> dest)
                finishLatch()(dest).await();
            }
        }
    }
}
```

Figure 7. Implementation example of send-receive operation in X10

Acknowledgments

We would like to thank the members of X10 project in IBM T. J. Watson Research Center and IBM Research - Tokyo, for their various suggestions and valuable comments. This material is based upon work supported by the Department of Energy under award DE-FOA-0000619.

References

[1] Center for Exascale Simulation of Advanced Reactors (CESAR), http://cesar.mcs.anl.gov/

[2] Exascale Co-design Center for Materials in Extreme Environments (ExMatEx), http://www.exmatex.org/

[3] Center for Exascale Simulation of Combustion in Turbulence (ExaCT), http://exactcodesign.org/

[4] Message Passing Interface Forum, http://www.mpi-forum.org/

[5] The OpenMP API Specification for Parallel Programming, http://openmp.org/wp/

[6] The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl/

```
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);
}

void stats(void) {
    RankReduceData sBuf[numOfT], rBuf[numOfT];
    minRankDouble(sBuf, rBuf, numOfT);
    ...
}
```

Figure 8. Implementation example of MPI_MINLOC in C

```
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);
        }
    }
}

public class Perf {
    val comm:Communicator;
    val sBuf = new Rail[Communicator.RankReducedData](numOfT);
    val rBuf = new Rail[Communicator.RankReducedData](numOfT);

    public def this(comm:Communicator) {
        this.comm = comm;
    }

    def stats():void {
        comm.minRankDouble(sBuf, rBuf, numOfT);
        ...
    }
}
```

Figure 9. Implementation example of MPI_MINLOC in X10

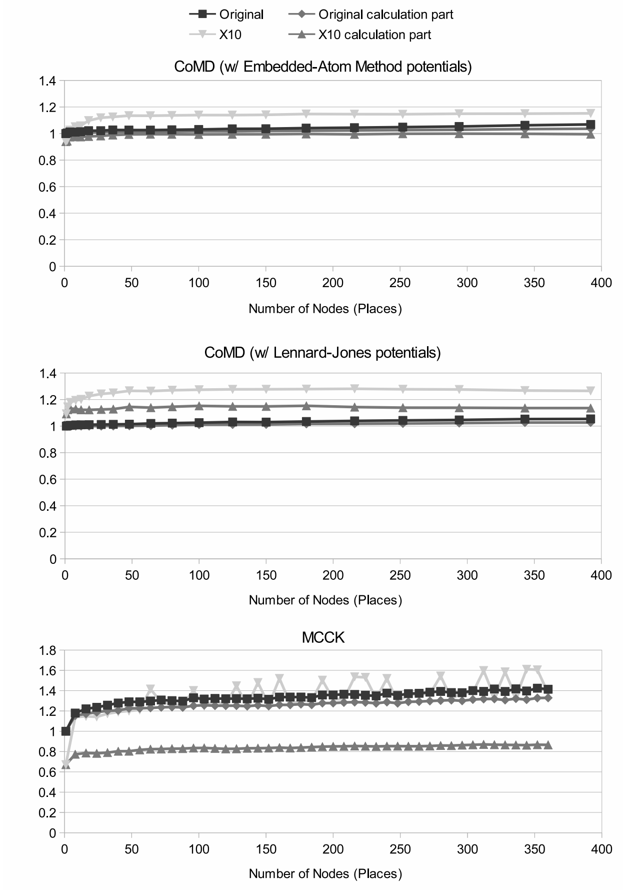

Figure 10. Weak scaling performance of applications (elapsed time relative to the original on 1 place)

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

[8] CoMD GitHub, https://github.com/exmatex/CoMD

[9] Monte Carlo Communication Kernel (MCCK), https://cesar.mcs.anl.gov/content/software/neutronics

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

[11] CoMD ported onto X10, http://svn.code.sf.net/p/x10/code/applications/trunk/CoMD/

[12] MCCK ported onto X10, http://svn.code.sf.net/p/x10/code/applications/trunk/MCCK/

[13] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application, Proceedings of 2013 IEEE 27th International Symposium on Parallel & Distributed Processing, pp. 919-932, 2013.

[14] Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH), https://codesign.llnl.gov/lulesh.php

[15] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language, International Journal of High Performance Computing Applications, Vol. 21, No. 3, pp. 291-312, Aug. 2007.

[16] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++, in Proceedings of OOPSLA ’93, pp. 91-108, September 1993.

[17] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 11, 2011.

[18] E. A. Luke and T. George. Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis, Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming, Vol. 15, No. 03, pp. 477-502, 2005.

[19] W.-Y. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, PACT ’05, pp. 267-278, 2005.

[20] D. Chavarria-Miranda and J. Mellor-Crummey. Effective Communication Coalescing for Data-Parallel Applications. Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 14-25, 2005.

[21] M. Gupta, E. Schonberg, and H. Srinivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. IEEE Transactions on Parallel and Distributed Systems, 7:689-704, 1996.

[22] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick. Automatic nonblocking communication for partitioned global address space programs. Proceedings of the 21st annual international conference on Supercomputing (ICS ’07), pp. 158-167.

[23] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT ’04, pp. 29-40.

[24] M. Alvanos, M. Farreras, E. Tiotto, J. N. Amaral, and X. Martorell. Improving communications in PGAS environments: static and dynamic coalescing in UPC, Proceedings of the 27th international ACM conference on International conference on supercomputing ICS’13, pp. 129-138, 2013.

IBM, Power, and POWER7 are the trademarks of International Business Machines Corporation in the United States, other countries, or both. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.