2011 IEEE International Parallel & Distributed Processing Symposium

Communication Optimizations for Distributed-Memory X10 Programs

Rajkishore Barik
Intel Corporation, Santa Clara, CA
rajkishore.barik@intel.com

Jisheng Zhao
Rice University, Houston, TX
jisheng.zhao@rice.edu

David Grove
IBM T.J Watson Research Center, NY
groved@us.ibm.com

Igor Peshansky
IBM T.J Watson Research Center, NY
igorp@us.ibm.com

Zoran Budimlić
Rice University, Houston, TX
zoran@rice.edu

Vivek Sarkar
Rice University, Houston, TX
vsarkar@rice.edu

***Abstract*—X10 is a new object-oriented PGAS (Partitioned Global Address Space) programming language with support for distributed asynchronous dynamic parallelism that goes beyond past SPMD message-passing models such as MPI and SPMD PGAS models such as UPC and Co-Array Fortran. The concurrency constructs in X10 make it possible to express complex computation and communication structures with higher productivity than other distributed-memory programming models. However, this productivity often comes at the cost of high performance overhead when the language is used in its full generality.**

This paper introduces high-level compiler optimizations and transformations to reduce communication and synchronization overheads in distributed-memory implementations of X10 programs. Specifically, we focus on locality optimizations such as scalar replacement and task localization, combined with supporting transformations such as loop distribution, scalar expansion, loop tiling, and loop splitting. We have completed a prototype implementation of these high-level optimizations, and performed a performance evaluation that shows significant improvements in performance, scalability, communication volume and number of tasks. We evaluated the communication optimizations on three platforms: a 128-node BlueGene/P cluster, a 32-node Nehalem cluster, and a 16-node Power7 cluster. On the BlueGene/P cluster, we observed a maximum performance improvement of 31.46× relative to the unoptimized case (for the `MolDyn` benchmark). On the Nehalem cluster, we observed a maximum performance improvement of 3.01× (for the `NQueens` benchmark) and on the Power7 cluster, we observed a maximum performance improvement of 2.73× (for the `MolDyn` benchmark). In addition, there was no case in which the optimized code was slower than the unoptimized case. We also believe that the optimizations presented in this paper will be necessary for any high-productivity PGAS language based on modern object-oriented principles, that is designed for execution on future Extreme Scale systems that place a high premium on locality improvement for performance and energy efficiency.

I. Introduction

Computer systems anticipated in the 2015 – 2020 timeframe are referred to as *Extreme Scale* [21] because they will be built using *massive multi-core processors* with hundreds of cores per chip. These systems pose new critical concurrency and energy efficiency challenges for software. From an application viewpoint, the key challenges are the ability to express all of the intrinsic parallelism and locality in a portable manner, while ensuring that this expression can be efficiently mapped on to Extreme Scale systems with processors that exhibit high variability and require high degrees of locality for performance and energy efficiency.

X10 [23] is a new object-oriented partitioned global address space (PGAS) programming language that is designed for future parallel computing platforms including Extreme Scale systems. It provides first-class support for distributing data and computation, as well as for creating dynamic tasks in a distributed-memory context, unlike past PGAS models (such as UPC [9] and Co-Array Fortran [20]) that only supported bulk-synchronous SPMD execution models. The concurrency constructs in X10 (such as `async`, `finish`, `at`, `ateach`, `atomic`, places, distributions) make it possible to express complex computation and communication structures with higher productivity than past distributed-memory programming models. However, this productivity often raises new challenges for code generation and optimization.

This paper introduces high-level compiler optimizations and transformations to reduce communication and synchronization overheads in distributed-memory implementations of X10 programs. The generality of X10 is manifested in its *distributed object model* which includes objects, structs, and closures. A key source of *communication overhead* relates to the serialization that is performed on an object and the subgraph of objects that it can reach, which directly impacts the volume of communicated data. A key source of *synchronization overhead* arises from lightweight tasks that cross place boundaries. Given this context, we focus on *locality optimizations* including *scalar replacement* and *task localization*, combined with supporting transformations such as loop distribution, scalar expansion, loop tiling, and loop splitting.

Using our prototype implementation we evaluated the performance of five programs (three benchmarks and two real-world applications) on three platforms: (1) a 128-node BlueGene/P cluster that is part of a 4096-node system; (2) a 32-node Nehalem cluster with Infiniband interconnect that

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.105

is part of a 90-node system; and (3) a 16-node Power7 cluster with Infiniband interconnect that is part of a 18-node system. On the BlueGene/P cluster, we observed a maximum performance improvement of 31.46× relative to the unoptimized case (for the `MolDyn` benchmark). On the Nehalem cluster, we observed a maximum performance improvement of 3.01× (for the `NQueens` benchmark) and on the Power7 cluster, we observed a maximum performance improvement of 2.73× (for the `MolDyn` benchmark). There was no case in which the optimized code was slower than the unoptimized case. Additionally, the experimental results show that our optimizations produce significant improvements in performance, scalability, communication volume and number of remotely spawned tasks.

The rest of the paper is organized as follows. Section II summarizes the X10 language constructs, compiler, and runtime. In Section III, we present a motivating example to illustrate the effectiveness of our approach. The communication optimizations and their soundness in the presence of exceptions are described in Section IV. Experimental evaluations are presented in Section V. We discuss related work in Section VI and conclude in Section VII.

II. X10

X10 is an object-oriented Partitioned Global Address Space (PGAS) language. The partitioned global address space is reified in X10 by the concept of a *place*. A computation in X10 may span multiple places. Objects residing in one place may contain references to objects residing in other places. However, X10 enforces a strong locality property: it is not permissible to access an object's mutable state through a remote reference to that object[1]. Therefore computations must sometimes "shift" from one place to another to access the data they need. When this happens, the compiler and runtime system must somehow communicate data and control information from one place to another. This section provides background information on the X10 2.0 programming language constructs and some implementation details of X10 2.0.6 relevant to understanding this cross-place communication. We focus on the subset of X10 that includes the following constructs: places, regions, distributions, `async`, `at`, `finish`, `ateach`, and `atomic`. For a more complete description of X10, please refer to the X10 language specification [23].

A computation in X10 consists of one or more asynchronous *activities* (light-weight tasks). A new activity is created by the statement `async S`. To synchronize activities, X10 provides the statement `finish S`. An activity that executes a finish statement will not execute the statement after the finish until all activities spawned within the finish's body have terminated.

Every activity executes in a single `Place` (address space). While executing in this place, it may freely access any object that also resides in the place. The fundamental X10 construct for "place-shifting" is `at (p) S`. An `at` statement shifts execution of the current activity from the current place to place p[2]. In X10 2.0, an `at` statement is implemented by the X10 compiler and runtime system by translation to an active message. The message identifier encodes the source code location (the code to execute at the target place when it receives the message). The message payload is a serialized form of the data necessary to execute the body of the `at`. To determine the data needed, the compiler analyzes `S` and identifies any upwardly exposed variables (variables referenced in `S` that are defined in its lexically enclosing environment). The compiler then generates code to serialize the values contained by each of these variables into a communication buffer. On the remote side, when the message is received, the runtime system de-serializes the values from the communication buffer, initializes the variables of `S` to refer to them, and then executes `S`. At the sending place, the activity that executes the `at` statement is blocked until it is notified that `S` has completed.

Exactly what data is serialized at an `at` is a function of X10's distributed object model. In X10 2.0, there are three kinds of values: objects, structs, and functions (closures). When an object is serialized, the subset of its instance fields that have been declared as `global` are serialized. Only `val` (immutable) fields may be declared as global. To enable later access to the non-global instance fields of the object, when the object is de-serialized in the destination place it will contain additional information (a remote reference to the original object) that can be used in subsequent `at` statements to return to the home location of the original object and access its non-global fields. It will also contain copies of all of the original object's global fields. Since these fields are immutable, they can be safely accessed in the remote location. When structs and functions are serialized, all of their data members are serialized (all instance fields of these types are implicitly global). When serializing a global instance field, the value contained by the field is recursively serialized. Thus when communicating the data for an `at` statement, for each upwardly exposed variable, the transitive closure of the object graph reachable by global instance fields is serialized.

In X10 2.0, the construct `async (p) S` is a syntactic sugar for `at (p) async S` and is implemented accordingly. X10 also provides the `ateach` construct as a convenient way of spawning multiple activities across a set of places. The statement `ateach (p in dist) S`

[1]The 1.7, 2.0, and 2.1 versions of X10 have had slightly different distributed object models, but this fundamental locality property is true of all three versions of the language. Mutable state can only be accessed in its home location.

[2]X10 also supports `at` expressions such as `at (p) E` and `at (p) S; E` that facilitate the return of a value from a remote computation.

is expanded by the front-end of the X10 compiler into an equivalent statement `for (p in dist.region) at (dist[p]) async S` and is implemented accordingly.

A core part of the functionality provided by the X10 class libraries is support for k-dimensional arrays (both single place and distributed). A `Point` of rank k represents an element in a k-dimensional Cartesian space with integer value coordinates. A `Region` is an ordered set of points, all of the same rank. An `Array` in X10 is defined over a `Region`; the `Array` maps every `Point` in its defining region to a corresponding data element. `Array` is a single-place construct; the `Array` object and all of its backing storage are allocated in a single `Place` and can only be accessed by activities executing in that `Place`. In X10 2.0.x, the class library also provides the `Rail` and `ValRail` classes, which are specializations of `Array` for the common case of 1-dimensional, dense, zero-based arrays. Because a `ValRail` is immutable, all of its elements will be serialized between places; in contrast when a `Rail` is serialized only a remote reference to the object is created (the elements are not serialized).

A `Dist` (distribution) maps each `Point` in a `Region` to a `Place`. A `DistArray` (distributed array) is defined over a `Dist`; the `DistArray` maps every `Point` in the `Region` of its defining `Dist` to a data element. Several simple distributions such as *unique*, *block*, and *block-cyclic* are built into the class libraries; user-defined distributions can also be used to define instance of `DistArray`. Unlike `Array`, `DistArray`'s backing storage is distributed among all of the `Place`s that are included in its `Dist`. A data element in a `DistArray` can only be accessed by activities that are executing in the `Place` to which the `Dist` maps the corresponding `Point`.

A lower-level facility provided by the class libraries is that of a `PlaceLocalHandle`. A `PlaceLocalHandle` in X10 fills a similar role as the Shared Variable Directory (SVD) in IBM's UPC implementation [5]. Namely, it provides a unique id that can be efficiently resolved to a unique local piece of storage at each `Place`. One important use of `PlaceLocalHandle` is to implement the backing storage for `DistArray`. Each instance of a `DistArray` internally has a single instance of a `PlaceLocalHandle` which it uses to find the place-local storage used as the backing data store for its data elements. Programs may also use `PlaceLocalHandle` directly, and several of our optimizations work by automatically transforming programs to introduce additional `PlaceLocalHandle`'s.

The X10 compiler infrastructure is depicted in Figure 1. This compilation framework is composed of two parts: an AST-based front-end that parses X10 source code and performs AST based program transformation; Native/Java backends that translate the X10 AST into C++/Java code and invokes the post compilation process that uses C/C++ compiler to build executable binary or javac compiler to

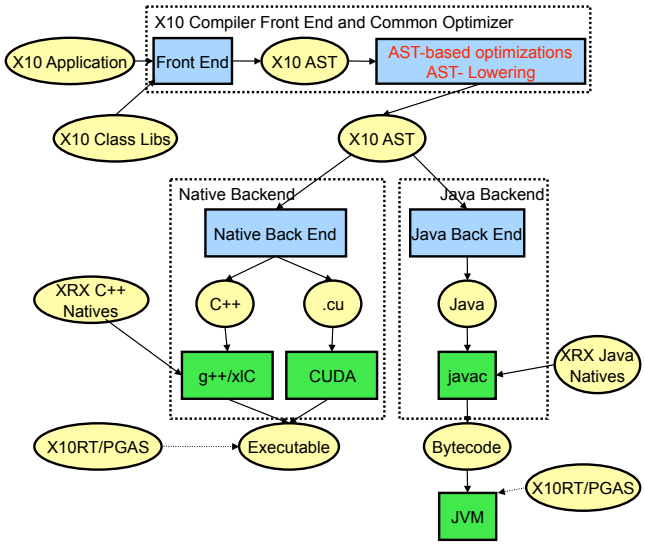

Figure 1. X10 Compiler Structure

build bytecode. Both binary code and Java bytecode need to co-operate with the X10 runtime system (X10RT) that manages task scheduling and message passing on the distributed memory system.

III. Motivating Example

Figure 2 (Original Code) presents a fragment of the distributed X10 version of the JGF *MolDyn* benchmark [17]. It creates an array of `Particle` objects, *i.e.,* `particleTable`, using the distributed array library interface of X10. The `particleTable` is distributed across *places* using a *block* distribution that is created using `Dist.makeBlock()`. The loop nest visits each `Particle` at a place using the j loop and for each such `Particle`, it traverses the remaining `Particle`s to compute pairwise force using the k loop. The `at` construct in the k-loop fetches the `Particle` objects from various places.

Even though the code in Figure 2 (Original Code) is succinct, it offers several opportunities for optimizing communication and synchronization overhead. First, the k loop nest creates unnecessary parallel tasks and their synchronization for fetching local data. Second, while the loop nest executes concurrently across places, it executes sequentially within a place due to the synchronization imposed by the `at` construct; some of these synchronizations are unnecessary and can be eliminated using compiler based transformations described in this paper.

Figure 2 (Optimized Code) presents the transformed code for (Original Code) after applying the following communication optimizations:

- First, *loop splitting* is applied to the k loop which separates local and remote communications while preserving the original program semantics. The local communications performed by the $k1$ loop can then *inline* the `at` construct and eliminate parallel task creation and

```
struct commBuf {
 x:double;y:double;z:double;
 def this(x:double,y:double,z:double) {
  this.x=x;this.y=y;this.z=z; } }
...
val dist = Dist.makeBlock([0..M-1]);
val particleTable = DistArray.make(dist, ...)(M);
val slice = M/Place.MAX_PLACES;
...
finish ateach (p in Dist.makeUnique()) {
  val i_ = here.id;
    for (var j:Int=i_*slice; j<(i_+1)*slice; j++) {
      for (var k:Int=j+1; k<M; k++) {
        val res = at (particleTable.dist(k)) {
          val tmpP = particleTable(k);
          val tx = tmpP.x, ty = tmpP.y, tz = tmpP.z;
          commBuf(tx, ty, tz)
        };
        rxcoord=res.x; rycoord=res.y; rzcoord=res.z;
      ...  } } }
```

Original Code

```
...
finish ateach (p in Dist.makeUnique()) {
  val commBufs:Rail[commBuf]=Rail.make[commBuf](slice);
  val i_ = here.id;
  for (var j:Int=i_*slice;j<(i_+1)*slice;j++) {
    // LOCAL
    for (var k1:Int=j+1;k1¡M && k1¡(i_+1)*slice;k1++) {
      // the following at is for  local place-shifting
      // and is unnecessary in this case
      val res = at (particleTable.dist(k1)) {
         val tmpP = particleTable(k1);
         val tx = tmpP.x, ty = tmpP.y,  tz = tmpP.z;
         commBuf(tx, ty, tz) };
     rxcoord = res.x; rycoord = res.y; rzcoord = res.z;
   ... } }
   // REMOTE
   val startPlace = i_+1;
   for (var k2:Int=startPlace;k2¡Place.MAX_PLACES;k2++) {
     finish at (places(k2)) async {
        for (k3=slice*k2;k3¡slice*(k2+1);k3++) {
          val tmpP = particleTable(k3);
          val tx = tmpP.x, ty = tmpP.y, tz = tmpP.z;
          val ind = k3 % slice as int;
          at (commBufs) async
            commBufs(ind) = commBuf(tx, ty, tz);
     } }
     for (var j:Int=i_ * slice;j<(i_ +1)*slice;j++) {
       for (k3=slice*k2;k3<slice*(k2+1);k3++) {
         val ind = k3 % slice as int;
         val cb = commBufs(ind);
         rxcoord = cb.x; rycoord = cb.y; rzcoord = cb.z;
       ... } } } }
```

Optimized Code

Figure 2. `MolDyn` example demonstrating the benefits of communication optimizations. We assume that the `particleTable` is *block* distributed across MAX_PLACES number of places. For simplicity sake, let us assume that the number of particles, M, is a multiple of MAX_PLACES (this is not a requirement). The `Original Code` is the unoptimized code for the `MolDyn` benchmark. The `Optimized Code` is obtained from `Original Code` by a series of compiler transformations described in this paper: (1) *loop splitting* of k loop for separating local ($k1$) and remote ($k2$) communications; (2) *strip-mine* and *scalar expansion* of remote loop $k2$ to parallelize the loop iterations and avoid fine-grained synchronization overhead; (3) *loop distribution* to distribute j loop to produce final code in `Optimized Code`. The transformations are described in detail in an expanded version of this paper in [3].

synchronization overheads. This is only valid if the `at` construct does not throw any exceptions, which is the case in this example. We discuss in Section IV-E how these transformations can be performed in the presence of exceptions.

- Second, the remote loop created after loop splitting is sequential due to the "place-shifting" construct `at`. This incurs synchronization overhead for every iteration of the remote loop. Since the body of the `at` construct only fetches data from remote places (and is free of loop-carried dependencies and side-effects), we can easily parallelize the loop by scalar expanding res to *commBufs*. For efficient space usage, we strip-mine the remote loop at a place-level using loop $k2$. Such a transformation results in: (a) parallelizing the loop nest; and (b) reducing the synchronization overhead to exactly one per place instead of every iteration in the original program.
- Finally, the j loop is distributed to both the $k1$ and $k2$ loop nests. This transformation removes additional overheads of parallel task creation and synchronization.

IV. Communication Optimization

In this section, we introduce compiler optimizations that reduce the communication overhead of distributed-memory X10 programs. As mentioned earlier, we focus on the subset of X10 that includes the following constructs: places, regions, distributions, `async`, `at`, `finish`, `ateach`, and `atomic`. The sources of communication overhead include: 1) the size of transferred data; 2) the number of communication operations, (*i.e.,* `async` and `at`); and 3) the number of synchronization operations (*i.e.,* `finish`). For 1), our approach is to apply *scalar replacement* and *object splitting* transformations to reduce the size of transferred data by removing certain classes of redundant data communications. For 2) and 3), our approach is to apply several program transformations (focused on loops) to reduce the number of communication and synchronization operations performed.

A. Scalar Replacement

As introduced in Section II, X10 serializes immutable classes, structs, and functions (closures) when they need to be passed across places. These forms of serialization will be necessary by default in any high-productivity PGAS language based on modern object-oriented principles. However, unlike other models, an X10 implementation need not worry about scenarios where read/write accesses can be performed on the same location by activities in more than one place. Since the communicated values are immutable, the compiler is free to communicate only a *live* subset of the values designated for data transfer, *e.g.,* in some cases, only a field of an object or an element of an array needs to be communicated instead of the entire object or entire array.

Scalar replacement [7], [10] is a classical compiler trans-

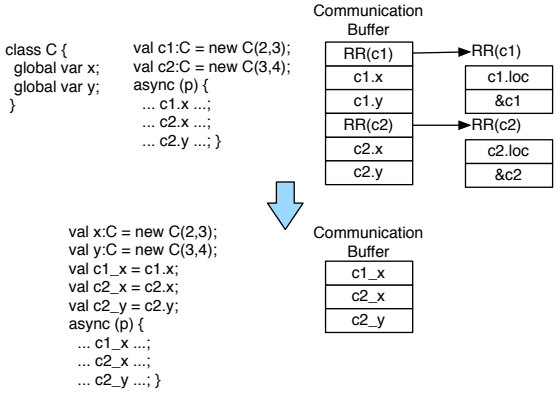

Figure 3. Communication optimization using scalar replacement for object field accesses

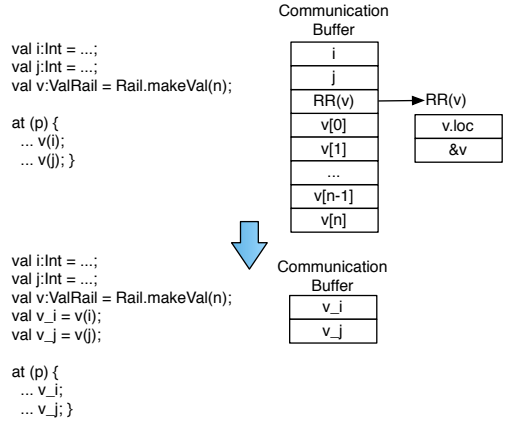

Figure 4. Communication optimization using scalar replacement for array accesses

formation used to replace accesses of object fields and array elements by accesses to scalar temporaries, thereby enabling more opportunities for register allocation. However, in this work, we use the scalar replacement transformation to eliminate redundant data communications in distributed-memory X10 programs, as explained in the following two scenarios.

The upper left part of Figure 3 shows a code fragment containing field accesses within a remote `async`. The `async`(p) statement creates an activity at the remote place p that performs memory load operations on object fields $c1.x$, $c2.x$, and $c2.y$. The unoptimized communication buffer that needs to be dispatched to place p along with the `async` is shown on the upper right; it consists of the remote reference (RR) handles for objects $c1$ and $c2$ including their fields x and y (shown on upper right corner of Figure 3). The remote reference handle captures the place of the object and its memory location at that place. In the optimized case, a compiler can scalar-replace the immutable field accesses $c1.x$, $c2.x$, and $c2.y$ in the `async`, reducing the size of the communication buffer (as shown in the lower right part of Figure 3). The transformed code after scalar replacement is shown in the lower left part. The optimized code does not need to send any remote reference handles.

In general, the precision of a scalar replacement transformation depends on the precision of the object alias analysis performed by the compiler. For the results reported in this paper, we used an efficient flow-insensitive alias analysis algorithm derived from the algorithms reported in [10] and [4]. Further, scalar replacement of array accesses requires that their bounds check operations are also redundant.

Figure 4 shows another code fragment performing array accesses inside a remote `async`. In general, array accesses $v(i)$ and $v(j)$ can access any element of the one dimensional array (`ValRail`) v. Without additional analysis across `async` boundaries, a compiler makes the worst-case assumption that the `async` can potentially access any element, forcing the communication of the whole array v

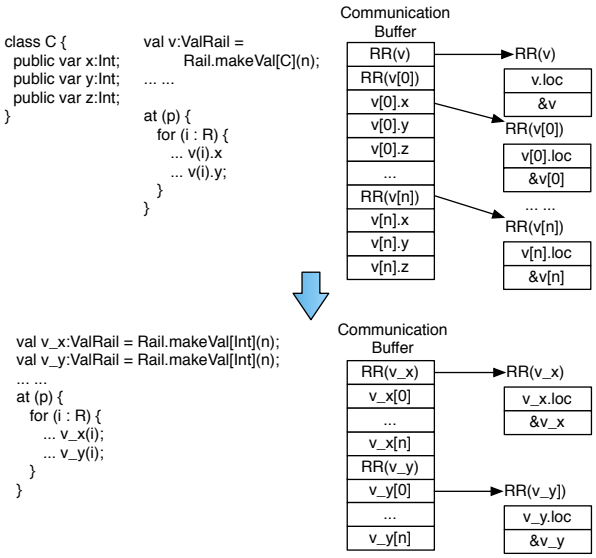

Figure 5. Communication optimization using class splitting

along with the `async` (as shown on the upper right of Figure 4). Scalar replacement of $v(i)$ and $v(j)$ reduces the communication buffer to two scalars, v_i and v_j (as shown on lower right).

B. Object Splitting

Consider the code example shown in Figure 5 that shows a class C with class fields: x, y, and z. The original program creates an array of objects of class C, however, only fields x and y are accessed in a remote task. Splitting the array [6] into separate arrays for fields x, y and z allows the compiler to eliminate the communication overhead of serializing the z field and the object header. This kind of opportunity occurs frequently in object-based applications.

C. Program Transformations

Our compiler includes several loop transformations to reduce the number of communication and synchronization

1. Scalar Replacement with Loop-Invariant Code Motion

```
class T ...
...
val A = Rail.make[T](r);
// region r is a subset of region R
for(point i in R) {
  val idx = f(i) % Places.MAX_PLACE;
  async (Place.places(idx)) g(A(idx));
}
```

$\Longrightarrow$

```
val A:ValRail[T] = ...;
// send once to each place and save in local storage
val sharedA =
 DistArray.make[ValRail[T]](Dist.makeUnique());
finish ateach (p in Place.places())
   sharedA(p) = A;
for(point i in R) {
  val idx = f(i) % Places.MAX_PLACE;
  async (Place.places(idx))
    g(sharedA(Place.places(idx))(idx));
}
```

2. Local Optimization for `async`:

```
for (point i in R) {
  // S has no block operations and is exception free
   async(places(f(i))) S;
}
```

$\Longrightarrow$

```
for (point i in R) {
  if (places(f(i)) == here)
    S;
  else
    async(places(f(i))) S;
}
```

3. Local Optimization with Loop Distribution:

```
val dist:Dist = ...;
A = DistArray.make[T](dist);
for (i in dist)
   async(A.dist(i)) {
     ... = A(i);
  }
```

$\Longrightarrow$

```
val dist_local = dist | here;
val dist_remote = dist - here;
for (point i in dist_local) {
  ... = A(i); }
for (point i in dist_remote) {
   async(A.dist(i)) {
     ... = A(i);
  }}
```

4. Synchronization Elimination using Scalar Expansion:

```
// A is a DistArray on distribution R
for (point i in R) {
   t = at(A.dist(i)) A(i);
  ... = t;
}
```

$\Longrightarrow$

```
// t_arr is scalar expanded version of t
t_arr = Rail.make[T](R.size());
finish for (point i in R) {
  at (A.dist(i)) async {
    val v = A(i);
    at (t_arr) async t_arr(i) = v;
  }}
for (point i in R) {
  ... = t_arr(i); }
```

5. Place-level Strip Mining

```
// A is a DistArray on a distribution R
for (point i in R) {
   t = at(A.dist(i)) A(i);
  ... = t;
}
```

$\Longrightarrow$

```
for (p in Place.places()) {
  val subdist = R | p;
  // t_arr is scalar expanded version of t
  val t_arr = Rail.make[T](subdist.size());
  finish at (p) async {
    var j:Int = 0;
    for (point i in subdist)) {
     val ind = j++; val v = A(i);
     at (t_arr) async t_arr(ind) = v;
    }}
  var j:Int = 0;
   for (point i in subdist)) {
    ... = t_arr(j); j++;
  }}
```

6. Parallel Reduction through Scalar Expansion

```
x = ...;
finish ateach (point in R) {
  ...
  val v = ...;
   at (x) { atomic x.val += v; }
  // no use of x.val
 }
  ... = x.val;
```

$\Longrightarrow$

```
x = ...;
x_tmp = DistArray.make[T](...);
finish ateach (point in R) {
  ...
  val v = ...;
  x_tmp(here.id) = v;
}
// Reduction across places
x.val = x_tmp.reduce(T.+, x.val);
  ... = x.val;
```

Figure 6. Communication optimization rules: Scalar Replacement, Local Optimization, Synchronization Elimination, Place-level Strip-mining, and Parallel reduction

operations performed. These transformations are summarized in Figure 6.

Scalar Replacement with Loop Invariant Code Motion: Figure 6-1. shows an example of a scalar replacement transformation combined with loop invariant code motion. The code on the left requires communicating the entire array `A` to the target place at every loop iteration. In the transformed code on the right, the array is first communicated to all the places and stored as a local copy in each place, thereby enabling later async's to only communicate the `idx` variable. This transformation also avoids the need for re-sending the array `A` while accessing the same place multiple times, as in the original case.

Local Optimization for async: Figure 6-2. shows an example of converting a local async into sequential code. The code on the left creates an async regardless of whether it is local or remote. The code on the right avoids the overhead of creating a local async when the target place happens to be local. In general, this transformation would require compiler analysis to prove that the statement `S` does not have any synchronization dependencies on other asyncs that may get spawned after the loop. However, in the subset of X10 that this paper focuses on, such dependencies are not possible, thus the compiler analysis is not necessary.

Local Optimization with Loop Distribution: Figure 6-3. shows how an array distribution can be partitioned into local and remote subsets. The code on the right first computes the local and remote distributions, then for all points in the local distribution it executes the computation (in this case, a simple assignment) sequentially on the local place, followed by the remote computation on the remote places.

Synchronization Elimination using Scalar Expansion: In the example in Figure 6-4., the compiler uses scalar expansion of variable `t` to enable parallel execution of tasks that would otherwise be sequentialized. The statement `t = at(p) S` involves a communication because statement `S` can be performed at remote place `p`. In this particular example (taken directly from the *MolDyn* benchmark) the remote computation is trivial (just returning the value of `A(i)`), but it can be arbitrarily complex in general. In the code on the left, the computation of `t` and its subsequent use are done in a sequential loop, therefore sequentializing all the remote task executions. The code on the right converts the `at` construct into a combination of `finish` and `async`, that allows the remote tasks to execute in parallel and assign their results to the scalar-expanded array `t_arr`. This results in eliminating the synchronization operations inherent in a place-shifting `at` statement. Note that the size of the scalar expanded array, `t_arr`, is proportional to the size of the original array, which can be expensive in general. The next transformation below can reduce the space overhead of compiler-generated array temporaries.

Place-level Strip Mining: The example in Figure 6-5. takes the example from Figure 6-4. one step further by first tiling the iteration space at the place level and then executing the tiles sequentially (using `finish` and `async` in order to reduce space overhead) while parallelizing the computations within a tile. Instead of creating an async for each iteration of the loop as in the code on the right of Figure 6-4., the code on the right of Figure 6-5. first finds a sub-distribution of `A` for each place, then creates a sequential task at each place to execute the place-local part of the computation in parallel. Compared to Figure 6-4, this case requires a space overhead proportional to the size of the local array within a place.

Parallel Reduction through Scalar Expansion: Figure 6-6. shows an example of how a reduction can be done in parallel through scalar expansion of the variable containing the result of the reduction. In the code on the left, all remote tasks are atomically updating the value of `x.val`, creating many tasks on the place that owns `x` just to update the value. In the code on the right, each remote task only updates its private copy (the element of the scalar-expanded `x_tmp` array), and the final reduction is performed by the X10 *reduce* operation.

D. Overall Communication Optimization Algorithm

Now we present an integrated algorithm based on the program transformations described above. The algorithm works on the Program Structure Tree (PST) [1] that is derived from the Abstract Syntax Tree (AST)-based intermediate representation (shown in Figure 1). Our transformations are applied before mapping the X10 AST to the C++ or Java backend. Each node in our PST represents loops, statements, and X10 parallel constructs such as `finish`, `async`, `at`, `ateach`, and `atomic`. The algorithm uses the PST as input and performs bottom-up traversal on the PST to apply the program transformations described above.

The main algorithm shown in Figure 7 consists of two parts: scalar replacement and loop-based program transformations. Scalar replacement is a pre-pass for redundant data transfer elimination. The second pass walks through the PST nodes in bottom-up order, identifies the potential data dependence and exceptions for each PST node, then applies the appropriate transformations listed in Figure 6.

E. Exception Semantics

In this section, we discuss how to perform the communication optimizations described earlier, while preserving the program semantics in the presence of exceptions. In X10, an uncaught exception thrown inside the body of an `async` construct terminates the current activity, and throws the exception to the immediately enclosing finish (IEF) operation [4]. Similarly, an exception thrown inside the body of an `at` expression/statement is thrown to its IEF since `at` is a sequential "place shifting" construct. The finish scope catches all the exceptions that are thrown inside its body

2. Local Optimization for `async`:

```
for (point i in R) {
  // S has no block operations
   async(places(f(i))) S;
}
```

$\Longrightarrow$

```
val Es = Rail.make[Exception](R.size());
for (point i in R) {
  if (places(f(i)) == here)
   try { S;
   } catch (Exception e){ Es(i) = e; }
  else
    async(places(f(i))) S;
}
foreach (point i in R)
  if (Es(i)) throw Es(i);
```

3. Local Optimization with Loop Distribution:

```
val dist:Dist = ...;
A = DistArray.make[T](dist);
for (i in dist)
  async(A.dist(i)) {
    ... = A(i);
  }
```

$\Longrightarrow$

```
val dist_local = dist | here;
val dist_remote = dist - here;
val Es =
  Rail.make[Exception](dist_local.size());
for (point i in dist_local) {
   try{ ... = A(i);
   } catch (Exception e){ Es[(i)=e; } }
for (point i in dist_remote) {
   async(A.dist(i)) {
    ... = A(i);
}}
foreach (point i in dist_local)
  if (Es(i)) throw Es(i);
```

4. Synchronization Elimination using Scalar Expansion:

```
// A is a DistArray on distribution R
// body of at expression is free of side effects
for (point i in R) {
  t = at(A.dist(i)) A(i);
  ... = t;
}
```

$\Longrightarrow$

```
// t_arr is scalar expanded version of t
t_arr = Rail.make[T](R.size());
Es = Rail.make[Exception](R.size());
finish for (point i in R) {
  at (A.dist(i)) async {
    var tmpE:Exception = null;
    try { val v = A(i); }
    catch (Exception e) { tmpE = e; }
    if (tmpE == null)
     at (t_arr) async t_arr(i) = v;
    else {
     val ex = tmpE; val ind = i;
     at (t_arr) async { t_arr(ind) = v;
                        Es(ind) = ex); }
}
for (point i in R) {
  if (!Es(i)) ... = t_arr(i);
  else throw Es(i); }
```

5. Place level Strip Mining

```
// A is a DistArray on a distribution R
for (point i in R) {
  t = at(A.dist(i)) A(i);
  ... = t;
}
```

$\Longrightarrow$

```
for (p in Place.places()) {
  val subdist = R | p;
  // t_arr is scalar expanded version of t
  val t_arr = Rail.make[T](subdist.size());
  val Es =
      Rail.make[Exception](subdist.size());
  finish at (p) async {
    var j:Int = 0;
    for (point i in subdist)) {
     var tmpE:Exception=null; val ind=j++;
     try { val v = A(i); }
     catch(Exception e) { tmpE = e; }
     if (tmpE == null)
      at (t_arr) async t_arr(ind) = v;
     else {
      val ex = tmpE;
      at (t_arr) async {t_arr(ind) = v;
                        Es(ind) = ex); } }
    } }
  var j:Int = 0;
  for (point i in subdist)) {
   if (!Es(j)) ... = t_arr(j++);
   else throw Es(j); }
  } }
```

Figure 8. Communication optimization rules in the presence of exceptions: Local Optimization, Synchronization Elimination, and Place-level Strip-mining. Note that `foreach (point i in R) S` is a syntactic sugar for `for (point i in R) async S`.

```
1  Procedure Main, Input : Root node P of method m's PST
2    ScalarReplace (P);
3    LoopTransform (P);
4  Procedure LoopTransform, Input : PST node P
5   children = GetChildren (P);
6   for each child, c ∈ children do
7       LoopTransform (c);
8   bool isTransformed = false;
9   if IsLoop(P) then
10      for each child c ∈ children do
            //Perform Case (1) of Figure 6
11          if isAsync(c) and ScalarReplaceWithLICM(P, c) then
12              return;
13          if isAsync(c) or isAt(c) then
                //Find heap variables accessed in c
14              hs = HeapAccess(c); //Check data dependence
15              if hasDependence(P, hs) == false then
16                  if LocalOptWithLoopDist(P, c) then
                        //Perform Case (3) of Figure 6
17                      isTransformed = true;
18                  else if SyncElimination(P, c) then
                        //Perform Case (4) of Figure 6
19                      isTransformed = true;
20                  else if AsyncCoalescing(P, c) then
                        //Perform Case (5) of Figure 6
21                      isTransformed = true;
22  else if isFinish(P) then
        //Perform Case (6) of Figure 6
23      ParellelReduction(P);
24  if isTransformed == false then
25      for each child, c ∈ children do
26          if isAsync(c) or isAt(c) then
                //Perform Case (2) of Figure 6
27              LocalOptimization(c);
```

Figure 7. High-level algorithm for performing communication optimizations. The function `ScalarReplace` performs scalar replacement transformation for both field accesses and array accesses.

and combines them into a `MultiException` structure. If unhandled, the combined structure is subsequently thrown to the enclosing finish scope after terminating the current activity. Since the `main` program of the application is always wrapped in an implicit finish scope, all uncaught exceptions are reported when the application terminates.

The transformations in Cases (1) and (6) in Figure 6 already preserve program semantics in the presence of exceptions. For the transformations described in Cases (2)-(5) in Figure 8, we present new transformations that preserve the exception semantics. The common approach for handling exceptions is to capture the exceptions thrown inside the statement body in an one dimensional array, Es, using `try-catch` blocks. These exceptions are subsequently thrown to the enclosing finish scope to preserve the semantics of the original program.

The transformations in Cases (4) and (5) of Figure 8 capture the exception surrounding the array access $A(i)$ and store the exception in Es that is located at the outermost place (where t_arr is also located). The storing of exceptions occurs in parallel for each activity in the innermost loop. Finally, the exceptions are scanned in the original order of the program to find the first index, i that threw the exception. Once such an exception is found, no further computation is performed, thereby preserving the semantics of the sequential execution of the original loop nest.

V. Experimental Evaluation

In this section we present an experimental evaluation of our communication optimizations implemented in the X10 2.0.6 compiler infrastructure. To demonstrate the benefits, we used the distributed MPI runtime version of the C++ backend. Our optimizations were implemented as AST to AST transformations before the code generation pass in the X10 compiler front-end.

A. Benchmarks and Applications

Since X10 is a new language, there is currently only a small number of X10 programs available for evaluation. To maximize the variety, we evaluated the performance of five programs (three benchmarks and two real-world applications) for this study: a) a straightforward implementation of the HPC Challenge `RandomAccess`[3] microbenchmark [15] (with a per-node local table size of 4096, and number of updates = (*local table size*) $\times$ (*number of places*) $\times$ 4); b) the `NQueens` benchmark from the X10 open source distribution [23] (using a 13 $\times$ 13 board size); c) the JGF `MolDyn` benchmark [17] ported to distributed X10 (using 6192 particles); d) the `FMM` application from the ANU chemistry simulation system [2]; and e) the `PME` application from the same system (both simulated with 20,000 atoms).

Subsequent releases of X10, in particular X10 2.1.1 released in January of 2011, have included significant performance and scalability improvements in the X10 standard class libraries. Unfortunately, we were not able to finish porting our compiler optimizations and benchmark suite to this latest version of X10 in time to be able to report new experimental results in this paper. This explains the significant differences in the baseline performance of the `FMM` and `PME` programs in this paper and those in [19].

B. Experimental Platforms

We obtained experimental results on three cluster platforms: (1) a 128-node (4 cards) BlueGene/P cluster that is part of a 4096-node (128 cards) system; (2) a 32-node Nehalem cluster with Infiniband interconnect that is part of a 90-node system; and (3) a 16-node Power7 cluster with Infiniband interconnect that is part of a 18-node system.

Each compute node in the BlueGene/P system has 4 850Mhz PowerPC 450 cores and 2 GB of memory. The nodes are connected in a torus network and run a custom OS kernel. We used IBM's XL compiler v9.0 to produce executable binary from the generated C++ code, and the PGAS_BGP library for message passing. Each node in the

[3]Our straightforward implementation of RandomAccess uses asyncs in the innermost loop instead of using an explicit RDMA API. As a result, the performance is not comparable to the those reported in [22]

Nehalem cluster is a dual Quadcore 2.4 GHz Intel Nehalem CPU with 12GB of memory, running RedHat Linux 5 OS. We used GCC 4.2 to produce executable binary from the generated C++ code, and OpenMPI v1.4.2 for message passing. Each node in the Power7 cluster is a eight quad-core 3.55 GHz IBM Power7 CPUs with 256GB of memory, running Red Hat Enterprise Linux 5.4. We used GCC 4.2 to produce executable binary from the generated C++ code and OpenMPI v1.4.1 for message passing. All results were obtained using the default value of `X10_NTHREADS=1` which only creates one worker thread per node.

C. Experimental Results

We report experimental results for two cases: 1) `UNOPT` – the baseline version without any communication optimization; 2) `OPT` – the optimized version that uses the techniques described in this paper. The data size was kept constant to evaluate the impact of strong scaling. Figure 9(a) reports the execution times in seconds for each benchmark under both `OPT` and `UNOPT` configurations on a BlueGene/P cluster. The X-axis shows the number of nodes (*i.e.,* X10 places) used. For the `UNOPT` case, the `MolDyn` benchmark ran out of memory with 2 and 4 places. `MolDyn` and `NQueens` benchmarks show significant performance benefits with `OPT` across the board, while other benchmarks also show performance benefits when the number of places is increased. For `MolDyn`, we obtain a maximum speedup of 31.46× and for `Nqueens`, we obtain a maximum speedup of 26.33×. The overall speedup across all the benchmarks (for a given number of places) is in the range of 1.52× to 31.46×.

On the Nehalem cluster, the `MolDyn` and `NQueens` benchmarks show significant performance benefits for all places, the `RandomAccess` and `ANU-FMM` benchmarks show performance benefits for larger number of places while `ANU-PME` shows a small performance improvement. For the `MolDyn` benchmark, we observe a maximum speedup of 2.99× and for `Nqueens`, we observe a maximum speedup of 3.01×. The overall speedup across all benchmarks (for a given number of places) is in the range of 1.22× to 3.01×.

Similar to Nehalem cluster, the `MolDyn` benchmark gives the best performance improvement on the Power7 cluster and the other benchmarks also show improvements. We observe a maximum speedup of 2.73× for `MolDyn`. The overall speedup across all benchmarks (for a given number of places) is in the range of 1.05× to 2.73×.

The number of bytes communicated[4] is reported in Table I. Using our optimizations, the bytes communicated for both `NQueens` and `MolDyn` are reduced significantly by `OPT`, thereby explaining the runtime benefits achieved. As a part of the communication optimization, we also reduced the number of asyncs created at remote places. The number of asyncs created across remote places is shown in Table II. Again, `NQueens` and `MolDyn` show significant improvement after applying the optimizations.

We do not provide any comparison between X10 and MPI programs in this paper, because it has been shown in past work that X10 is able to achieve performance comparable to MPI for many HPCC benchmarks [22]. For larger applications, it will be a tedious effort to rewrite parallel object-oriented X10 programs in MPI.

[4]This statistic is reported by the X10 runtime system which is platform independent, so the value is the same for all three cluster systems.

VI. Related Work

The Fortran D project [14], [13] describes analysis, optimization and code generation techniques for the Fortran D compiler. They employ communication optimizations such as *message vectorization* (aggregating communications at an outer loop level), *message coalescing* (combining communications for different arrays in a single message), and *message pipelining* (to hide the latencies of message send and receive operations). These techniques complement the communication optimizations introduced in this paper which are focused on reducing the number of bytes communicated and the number of remote tasks created in parallel object-oriented programs. There have been some compiler optimization frameworks that applied the techniques described above for Fortran D to array-based data-parallel programs written in languages such as HPF [11], [18], [8].

The UPC compiler and runtime work in [5] presented three key compiler optimizations: 1) eliminating branch instructions for integer based affinity expressions in upc_forall loops; 2) eliminating accesses to shared pointers proven by the compiler to be local; (3) using efficient messaging mechanism for read-modify-write. Item 2) is related to optimization scenarios 2 and 3 described in Figure 6. While their aim is to prove some array accesses as local to avoid representing them as fat-pointers, our work focuses on reducing the communication overhead by creating fewer number of asyncs. We extend their local optimizations to the loop-level as shown in Figure 6.

In [24], Chen et. al. introduce compiler optimizations for fine-grain communication in UPC using an extension of SSA form that supports both scalar variables and indirect memory references. The optimizations include: 1) eliminating redundant communication, 2) coalescing fine-grain communication, and 3) splitting the read/write phase to reduce unnecessary message passing.

Scalar replacement for load elimination [4] extends scalar replacement across both method calls and parallel constructs (such as `async`, `finish`, and `isolated` in Habanero-Java [12]), using a relaxed isolation consistency memory model. Optimizations presented in this paper focus on scalar replacement of immutable values in a distributed memory environment whereas their paper focused on scalar replacement for both mutable and immutable object fields in a shared memory environment. In addition, our paper presents several

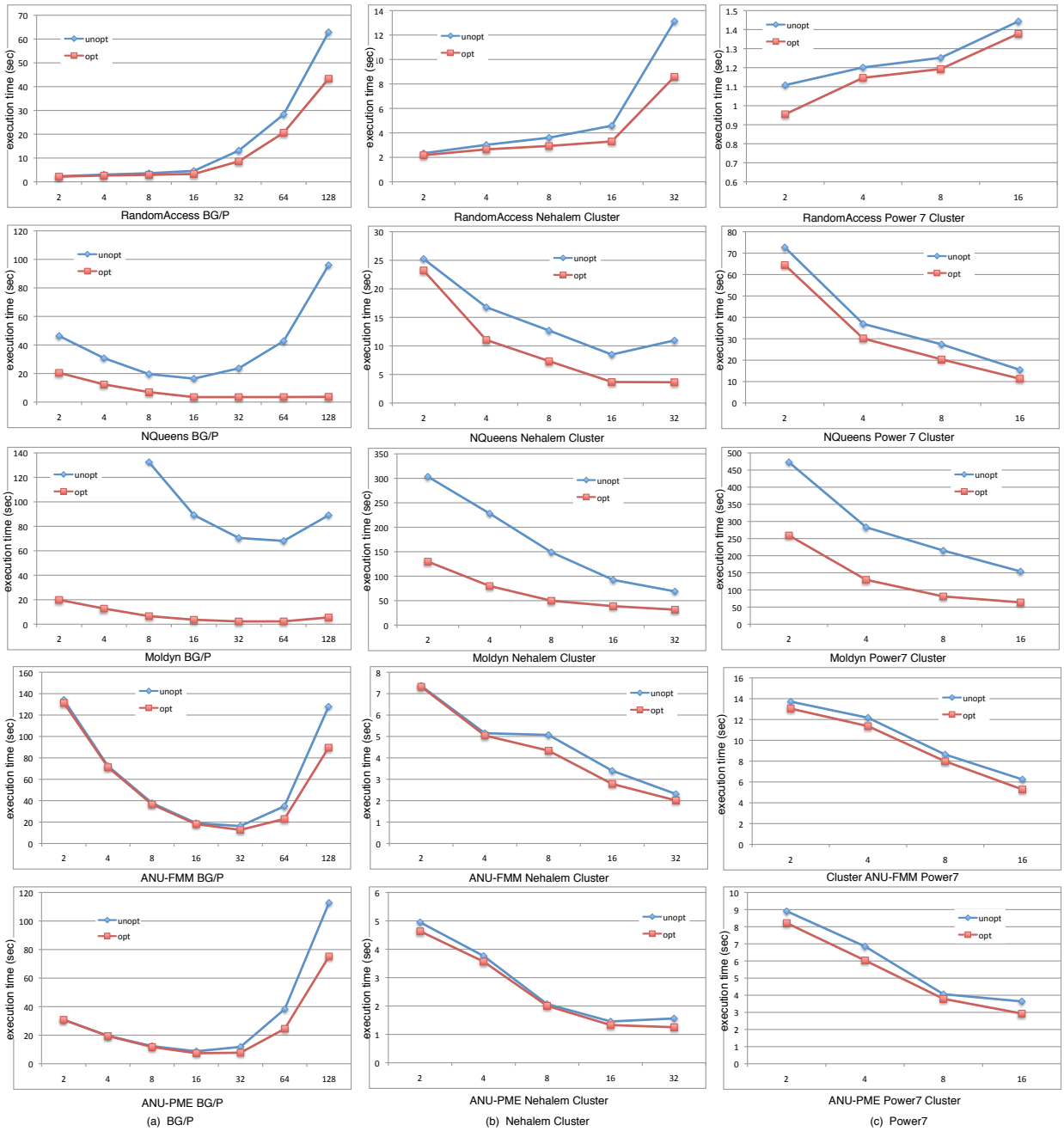

Figure 9. Comparison of execution time in seconds (y-axis) between optimized and unoptimized X10 programs on BG/P, Nehalem and Power7 cluster for varying number of nodes (x-axis).

Benchmarks		2	4	8	16	32	64	128
RandomAccess	unopt	2,256,856	9,660,964	355,32,854	131,797,506	500,538,822	1,943,077,198	7,643,344,770
	opt	915,870	2,857,120	6,875,010	15,042,700	31,534,622	63,856,368	130,392,532
NQueens	unopt	19,169,496	44,506,378	82,602,428	148,833,610	271,789,450	518,362,570	1,010,751,900
	opt	2,612	11,166	41,286	153,750	587,574	2,290,806	9,039,606
MolDyn	unopt	551,633,524	1,166,187,282	2,167,104,718	4,106,195,814	8,099,838,556	16,749,488,734	36,768,582,822
	opt	394,952	1,207,980	2,932,524	6,849,900	17,138,316	52,175,308	217,247,724
ANU.FMM	unopt	11,647,144	33,345,944	69,699,994	140,975,486	312,012,055	772,934,354	2,219,485,689
	opt	10,303,214	30,702,336	64,191,648	130,011,234	289,854,132	727,818,215	2,127,431,024
ANU.PME	unopt	24,388,118	61,949,648	106,236,886	188,299,146	370,993,106	831,975,294	833,652,654
	opt	23,277,968	59,160,890	101,252,764	177,328,274	348,954,582	788,977,118	799,426,694

Table I
NUMBER OF SERIALIZED BYTES COMMUNICATED ACROSS PLACES

Benchmarks		2	4	8	16	32	64	128
RandomAccess	unopt	65,540	131,080	262,160	524,320	1,048,640	2,097,280	4,194,560
	opt	65,542	131,084	262,168	524,336	1,048,672	2,097,344	4,194,688
NQueens	unopt	73,714	73,716	73,720	73,728	73,744	73,776	73,840
	opt	6	12	24	48	96	192	384
MolDyn	unopt	4,192,256	4,192,256	4,192,256	4,192,256	4,192,256	4,192,256	4,192,256
	opt	3,457	10,374	24,220	51,960	107,632	219,744	447,040
ANU.FMM	unopt	26,454	27,864	30,448	34,168	40,600	50,768	68,878
	opt	25,430	26,836	29,416	33,128	39,544	49,680	67,725
ANU.PME	unopt	809,262	809,790	810,441	811,291	813,250	816,381	820,985
	opt	808,695	809,171	809,537	810,116	810,712	811,822	813,375

Table II
NUMBER OF TASK SPAWNED ACROSS PLACES.

other loop transformations for reducing communication and synchronization overhead.

Chapel [16] is another new high-productivity language developed in the same timeframe as X10. Like X10, Chapel has a PGAS memory model with language-based notation for global arrays, global pointers, and locality exploitation. A key difference from X10 is that Chapel permits implicit accesses to remote locations, whereas X10 requires that all data access be place-local. However, despite this difference, many of the techniques presented in this paper can also be applied to Chapel programs.

VII. CONCLUSIONS AND FUTURE WORK

This paper introduced high-level compiler optimizations and transformations to reduce communication and synchronization overheads in distributed-memory implementations of X10 programs, paying close attention to the overheads inherent in dynamic task parallelism with a distributed object model. Using our prototype implementation of these high-level optimizations, we evaluated the performance of five programs on a Blue Gene/P cluster, a Nehalem, and a Power7 cluster. On the BlueGene/P cluster, we observed a maximum performance improvement of 31.46× relative to the unoptimized case (for the `MolDyn` benchmark). On the Nehalem cluster, we observed a maximum performance improvement of 3.01× (for the `NQueens` benchmark) and on the Power7 cluster, we observed a maximum performance improvement of 2.73× (for the `MolDyn` benchmark). There was no case in which the optimized code was slower than the unoptimized case. Additionally, the experimental results show that our optimizations produce significant improvements in performance, scalability, communication volume and number of remotely spawned tasks. We also believe that the optimizations presented in this paper will be necessary for any high-productivity PGAS language based on modern object-oriented principles, that is designed for execution on future Extreme Scale systems that place a high premium on locality improvement for performance and energy efficiency.

For future work, we will port our compiler optimizations to the latest X10 version (currently 2.1.2). As a number of scalability improvements were made to the X10 standard library in both the 2.1.1 and 2.1.2 releases, we expect to see a general reduction in overall serialization costs on the majority of the benchmarks studied in this paper. However, it is not clear if this will decrease or increase the relative effectiveness of our communication and synchronization optimizations. On the one hand there will be less "low hanging fruit" in the class library to optimize, but on the other hand those optimization opportunities that still exist may have a greater relative impact because of the improved baseline performance.

We would also like to implement our optimizations using an interprocedural analysis framework such as WALA, which may produce better performance results than our current and somewhat conservative implementation within the polyglot frontend of the compiler. Another interesting future direction is to extend the scalar replacement optimization for mutable values, which will require extending the compiler analysis and adding memory model consideration to the analysis.

REFERENCES

[1] Shivali Agarwal et al. May-happen-in-parallel analysis of X10 programs. In *PPoPP'07*, pages 183–193, New York, USA.

[2] ANU Computational Chemistry Applications. http://cs.anu.edu.au/∼Josh.Milthorpe/x10.html.

[3] R. Barik et al. Communication optimizations for distributed-memory x10 programs. Technical Report TR10-09, Department of Computer Science, Rice University, September 2010.

[4] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT'09*, North Carolina, 2009.

[5] Christopher Barton et al. Shared memory programming for large scale machines. In *PLDI '06*, Canada.

[6] Zoran Budimlić and Ken Kennedy. Optimizing Java - theory and practice. *Concurrency, Practice and Experience*, 9:445–463, 1997.

[7] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *PLDI '90*, pages 53–65, New York, NY, USA, 1990. ACM.

[8] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *PPoPP '05*, pages 14–25, New York, NY, 2005. ACM.

[9] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specification v1.1.1, October 2003.

[10] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS'00*, pages 155–174, 2000.

[11] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 7(7):689–704, 1996.

[12] Habanero Java. http://habanero.rice.edu/hj, Dec 2009.

[13] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *ICS'92*, July 1992.

[14] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM*, 35(8):66–80, 1992.

[15] HPC challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[16] Cray Inc. The Chapel language specification version 0.4. Technical report, Cray Inc., February 2005.

[17] The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[18] M. Kandemir et al. A global communication optimization technique based on data-flow analysis and linear algebra. In *PLDI'98*, 1998.

[19] Josh Milthorpe, V. Ganesh, Alistair P. Rendell, and David Grove. X10 as a parallel language for scientific computation: Practice and experience. In *25th IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[20] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, August 1998.

[21] Vivek Sarkar, William Harrod, and Allan E. Snavely. Software challenges in extreme scale systems. January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[22] X10 HPCC'09 tutorial. http://www.hpcchallenge.org/presentations/sc2009/hpcc09.pdf, October 2009.

[23] X10 programming language web site. http://x10.codehaus.org/, January 2010.

[24] Wei yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained UPC applications. In *PACT'05*, pages 267–278, 2005.