Retargetable Communication for Distributed Programs

Oren Freiberg
Microsoft
Email: aitnoren@cs.ucla.edu

Jens Palsberg
University of California, Los Angeles
Email: palsberg@ucla.edu

Mahdi Eslamimehr
Viewpoints Research Institute
Email: eslamimehr@ucla.edu

***Abstract*—The emergence of clusters of multi-core multiprocessors has created a challenge for software developers who use concurrency to gain performance. The challenge lies in the application's dependence on both the hardware and the deeply integrated communication infrastructure for performance improvements. This integration of the communication and parallelism in the user's application reduces flexibility by adding complexity when switching to different communication and parallel infrastructures. In this paper, we present a retargetable compiler framework for a subset of X10 that abstracts the hardware details, parallelism, and communication away from the application, allowing for portability and easier retargeting of the communication and parallelism. The retargetable compiler framework uses asynchronous computation and communication, as well as the concept of places to abstract away hardware details and to provide scalability. The framework offers performance, functionality, and flexibility because of our separation of tasks into layers and because of source code level serialization. To illustrate the ease of retargeting the communication and the patterns of parallelism, our framework is implemented with two different communication APIs (DUP and MPI-2) and two different patterns of parallelism (thread pooling and thread spawning). Retargeting the communication infrastructure using our framework required fewer code changes than changing the pattern of parallelism. The minimal code change needed to retarget these components offers developers a reasonable way to retarget without recompiling their application or sacrificing performance.**

I. Introduction

The increasing prevalence of multi-core, many-core, and distributed systems is pushing software developers to rely on concurrency and distributed computing to obtain peak performance [1]. Unfortunately, desired solutions in parallel computing often heavily depend on the underlying hardware (e.g., the number of processors per machine and the total number of machines). Furthermore, developers are obliged to support heterogeneous hardware setups in order to gain domain-specific performance increases. The heavy cost is a steep reduction in flexibility, since the communication mechanism for distributed computing is integrated deeply into the application [2], [3], [4]. Both Paalvast et al [2], [3] and Hiranandani et al. [4] suggest that changing the application's means of communication and parallelism is time consuming, difficult, and reduces code longevity. Because some benchmarks presented in our work perform better with specific patterns of parallelism and communication libraries, a reduction in flexibility could consequently hinder performance. Without flexibility, it is difficult to cater the communication or concurrency to the application. Further, a lack of flexibility can limit the portability and scalability of an application and prevent it from being able to run on different cluster configurations.

We address these issues with our retargetable X10 compiler framework [1] that offers the programmer the ability to retarget the communication backend and change the model of concurrency, all without recompiling the source application. Our retargetability is a result of our separation of tasks into three layers presented in Figure 1. The framework is split into the Application Layer, which provides access to the customized serialization generated for the user's application, the LocationSychronization Layer, which abstracts away the parallelism infrastructure and hardware from the application, and the Communication Layer, which abstracts the communication infrastructure from the application. The Application Layer is automatically generated by our retargetable compiler, which customizes the layer to the source application. This customization is one factor that aids our compiler in offering more features (like allowing any communication framework or parallel library to be plugged in) than the comparable IBM X10 solutions Not only are more features supported by our compiler but it offers comparable performance to the IBM X10-2.1.2 implementation.

II. The Challenges and the Current Landscape

We present an example of the difficulties of retargeting an MPI-2 program [3] in Figures 2, 3, and 4. In Figure 2, a node is receiving instructions via the `MPI_RECV` to spawn a thread, Thread1. On lines 2 and 3 in Figure 3, Thread1 is using an `MPI_SEND` to initiate parallelism by spawning two children to each execute the body of Figure 4. Thread1 then waits for its children to send back a finished message; waiting for the message simulates a barrier on lines 5 and 6. In Figure 4, ThreadVal is spawned to work on the val data it received via the `MPI_RECV`. Once the two ThreadVal threads are done executing they send a finished message to node 1 to notify Thread1 of their completion as seen in Figure 4 line 5. In the example, if we were to switch communication framework from MPI to DUP [5], a multi-stream based communication framework, it would require major restructuring. Naively switching all the `MPI_SEND` and `MPI_RECV` statements to the corresponding DUP send and receive statements would cause the program to execute incorrectly. Incorrect execution occurs because DUP messages are sent and received via file descriptors unlike MPI, and

[1]Our retargetable X10 compiler source with all benchmarks can be found at https://github.com/Mah-D/Retargetable-Compiler

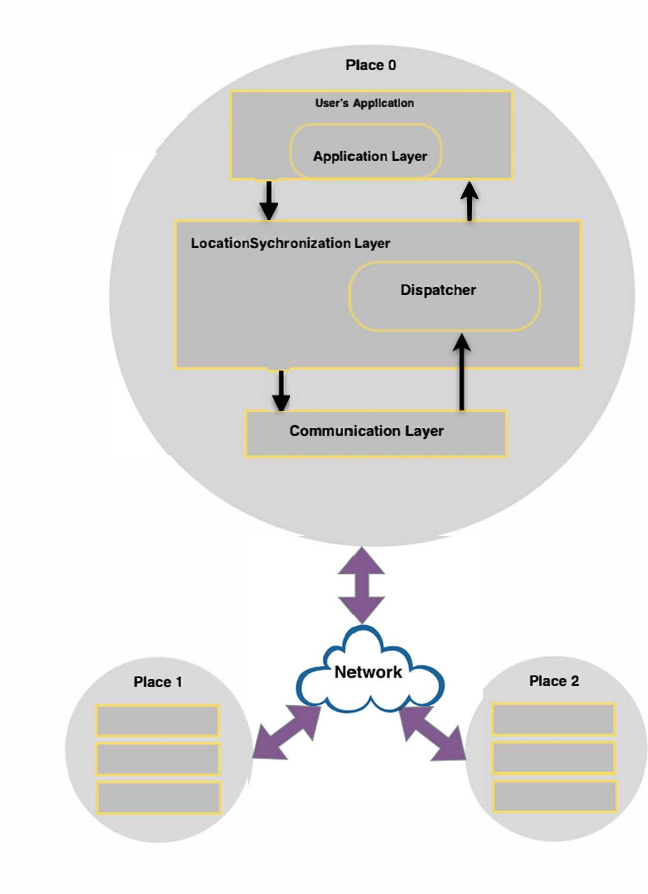

Fig. 1. The framework presented on three places

```
Node 1: receiving instructions to execute
Thread1
1: int p= COMM_WORLD.Get_rank ( );
2: struct params;
3: MPI_RECV(params,SIZE,Struct,p-1,Thread1)
//spawn a thread to run Program.thread1(params)
4: Program.thread1(params);
//Join on threads
```

Fig. 2. Thread1: at node 1

```
struct val;
MPI_SEND(val,SIZE,Struct,p+1,ThreadVal)
MPI_SEND(val,SIZE,Struct,p+1,ThreadVal)
//Waiting for child threads to finish execution
MPI_RECV(ack,SIZE,int,p+1,FinishVal)
MPI_RECV(ack,SIZE,int,p+1,FinishVal)
```

Fig. 3. Thread2 : at node 2

without a universal write and read lock per node, messages can get overwritten or corrupted. The complexity is exacerbated by the dependence of parallelism controls on the communication infrastructure, making the synchronization a difficult task and often requiring restructuring of the whole application. *This hardship illustrates the need for a more modular communication infrastructure that is not tightly coupled with the application, parallelism controls, or underlying hardware.* Our framework addresses this problem by providing portability, retargetable communication, and retargetable parallelism, all while maintaining performance.

```
MPI_RECV(ack,SIZE,int,p+1,FinishVal)
//spawn thread to run Program.threadVal
Program.threadVal(Val);
//Sending Thread1 a finish message
Send (1,SIZE,int,p-1,FinishVal)
```

Fig. 4. ThreadVal : at node 2

Our framework is designed to support single place, multiple data (SPMD) computation models where parallelism is expressed through async, finish, and place, key parallel constructs in X10 [6]. These constructs allow the parallel implementation and hardware details to be abstracted out of the source application and into the LocationSychronization Layer. A challenge with modularizing the parallelism infrastructure for a distributed system is that parallelism can be tied to the communication infrastructure because messages are needed for barriers across nodes and for task level parallelism. While challenging, it is important to have the flexibility to change the patterns of parallelism because different patterns can result in performance gains. These performance gains are illustrated in our performance results where the thread pool pattern outperformed thread spawning for both our Stream and Series benchmarks. Although improvements were noticed for both the Series and Stream benchmarks, the thread pool model is not always best. The Linpack benchmark, for example, does not execute correctly while using the thread pool pattern because it runs out of memory on even small input sizes due to the large number of threads needed to be preallocated. However, Linpack executes correctly when utilizing the thread spawning pattern, an example of why it is important to be able to switch to different patterns of parallelism.

We utilize X10-1.5 as our source language to leverage the benefits of X10, a place, hardware and communication agnostic source level language. Our work is related to the IBM X10 runtimes, which also relies upon variations of the X10 language as its source language. Flat-X10, X10-1.5 and X10-2.1.2 represent the breadth of research in this area. In Figure 5 we compare the features supported by the three IBM runtimes with our X10 retargetable compiler. Bikshandi et al. [7] presents an implementation of Flat-X10, a subset of X10 designed for distributed execution. Flat-X10 is designed for high performance distributed execution of the X10 language, but has several limitations. Flat-X10 is limited to the LAPI API for communication, lacks support for both arbitrary distributions and for arbitrary nesting of classes and arrays. Further, Flat-X10 does not support nesting of async and finish.

The IBM X10-1.5 implementation was built with a shared memory model allowing for execution on only one machine. The X10-1.5 implementation supports all properties of the original X10 language, including arbitrary distributions and the nesting of async and finish. It also supports the nesting of arbitrary classes and arrays but the difficulties with supporting

	X10-1.5	Flat-X10	X10-2.1.2	RC
shared memory execution	✓		✓	✓
distributed execution		✓	✓	✓
multiple communication libraries			✓	✓
arbitrary communication libraries				✓
switching communication requires no recompilation of source application				✓
retargeting patterns of parallelism				✓
nesting of async and finish	✓		✓	✓
nesting of classes and arrays	✓			✓
arbitrary distributions	✓			✓
control over place to node mapping				✓
garbage collectiom	✓		✓	✓

Fig. 5. A comparison of features supported by the three IBM runtimes and our retargetable compiler framework (RC)

this feature arise with distributed execution.

IBM's X10-2.1.2 implementation has added support for nested async and finish and added support for a subset of communication libraries. X10-2.1.2 does not support arbitrary communication libraries but only supports a subset that match their template. Further, the communication code is directly injected into the application, requiring the source application to be recompiled when switching communication libraries. X10-2.1.2 does not support arbitrary distributions or the nesting of classes and arrays. This means classes and arrays cannot contain any classes or arrays, reducing the programmer's flexibility. Another limitation of the X10-2.1.2 implementation is that the programmer has no control of which nodes are mapped to which place. This poses a particular problem when running on a heterogeneous cluster since two places (a place being a single operating system process) can get mapped to the slowest node or the node with least memory in the cluster. Our framework allows the programmer to specify a mapping of places to nodes to avoid such problems. Our framework also supports retargeting to arbitrary communication libraries and different patterns of parallelism because we separate these tasks into separate layers. The separation of tasks into layers provides an abstraction between the hardware, the application, the communication, and the parallelism. The result is a source application independent of the LocationSychronization Layer and Communication Layer, resulting in retargetability and longer code life cycles because changes in these layers require no recompilation of the source application.

Our framework is able to support arbitrary distributions and arbitrary nesting of classes and arrays because our Application Layer is customized to each solution. Classes and arrays which have classes and arrays nested in them must be serialized into a contiguous block of memory in order to support this. Our compiler automatically analyzes the source program and generates customized serialization functions for classes and arrays that are utilized within an *async*. Not only does our framework support a larger subset of features, but we also offer comparable performance to the X10-2.1.2 implementation. One reason the framework maintains performance is because it is compiling, it customizes the serialization to the source program. Without this customization, the application would either take a performance hit or remove flexibility from the programmer.

Our framework is not just limited to X10 and is usable by any language that has the same high level constructs, including Habanero-C, OpenMP, and Habanero-Java. This is because our separation of tasks into layers allows the source language to be independent of the LocationSychronization Layer and the Communication Layer. We demonstrate the ease of retargeting communication with three communication libraries: DUP, which utilizes stream based communication, and MPICH2 [8] and OpenMPI [9], two libraries which utilize point to point communication.

In summary, our paper makes several contributions:

- We provide a retargetable compiler framework that compiles X10 source to C, supports multi-threaded and distributed execution, and demonstrates retargetability through three communications libraries (DUP, OpenMPI, and MPICH2) and two different parallel patterns (thread spawning and thread pools),
- We offer a framework with three layers that provide a natural way to abstract the implementation of parallelism and communication from the application and that allow for the nesting of async and finish,
- We present benchmarks compiled and executed using our retargetable compiler framework to illustrate our comparable performance to the X10-2.1.2 implementation and to illustrate portability by running on two different clusters,
- We deliver support for nested classes, nested arrays and arbitrary distributions through a customized Application Layer.

III. X10-1.5

In this section, we provide an overview of PlasmaX10, our subset of X10-1.5. The X10 language is an explicitly parallel language that provides the programmer with a high degree of expressiveness and utilizes an Asynchronous Partitioned Global Address Spaces (APGAS) and an active messaging [10] model for computation. An APGAS model allows for an object at one place to point to an object that resides in the address space of a different place, which can manifest as accesses via an async. A place in X10 defines the address space or locality such that all concurrent code executing at the same place will share the same address space. A place can be thought of as a node, a processor or even a CPU. X10 is different from other models of parallel languages such as Titanium [11], which unlike X10 is limited to distributing arrays with

only a rectangular set of array indices. X10 allows for data distribution through multidimensional distributed arrays over an arbitrary set of indices.

```
(Statement) s ::=
  async ( id ){s}     parallelism
 |finish {s}          synchronization
```

Fig. 6. Parallelism and Synchronization constructs in X10

Our subset of X10, PlasmaX10, is synonymous with the programming language Java, except with thread and distributed execution expressed through high level concepts such as async, finish, and place. Although PlasmaX10 lacks both inheritance and abstract classes and is just a subset of X10, it is expressive enough to implement many distributed and shared memory benchmarks from both the Java Grande and HPC Challenge benchmarks. PlasmaX10 is very similar to a PGAS language that utilizes active messaging [10] . The similarity allows PlasmaX10 to express programs written in PGAS languages, which support shared memory or utilize message-passing libraries like UPC[12] and Titanium.

Parallelism, in X10, is expressed through async while synchronization is expressed through finish statements as seen in Figure 6. An async executes its body at a given place which is labeled id in Figure 6. Data declared final and utilized by the body of the async is transmitted to the place where the async will execute. Synchronization is expressed through finish statements, which block until all asyncs spawned inside its body terminate, including all recursively called asyncs. For example in Figure 7, an async is spawned inside of the finish block to execute a function at the next place. The async on line 5 will execute Program.thread1 at the given place while the finish waits at the end of the block for the async and all of its potential children asyncs to complete execution. The code in Figure 7 is a translation of our example code in Figure 3 to X10, except with only one thread being spawned.

A. PlasmaX10 to C

We built a compiler that translates PlasmaX10 to C and generates the Application Layer within the resulting C code. We compile to C to benefit from the language's tools, runtime support, optimized compiler, and flexibility to build for and respond to many different architectures. The framework generates all the code for the Application Layer including the customizable serialization code, freeing an X10 developer from writing any new code to use our framework. The compiler translates PlasmaX10 code found in Figure 6 to the C code presented in Figure 7, discussed in more detail in section IV-B. The example in Figure 6 and Figure 7 is initiating

```
1 final place p = ( here ) ;
2 final X x = ( new X ( ) ) ;
3 finish {
4    final T1 params = new T1(x);
5    async (p . next ( )) {
6       Program.thread1(params);
7 }}//end of async and finish
```

Fig. 7. X10 Sample Code

```
1  const place_t p = here();
2   /*finish*/ {
3    struct X x;
4    task_start_finish();
5    struct T1 params;
6    T1_T1( &params, x );
7    /* async */{
8      struct _struct_async async;
9      async.method = (ASYNC0) ;
10     async.size = sizeof(params);
11     async.params = (void *)(& params);
12     task_dispatch(a, place_next(p));
13     }//end of async
14   task_end_finish();
15 }//end of finish
```

Fig. 8. Translation of X10 to C

work at a new place and then waiting for its completion, analogous to Figure 4. The translation from X10 to C is done by replacing async, finish, and place statements with public method calls in the LocationSychronization Layer. The public method calls allow for parallelism and communication constructs to be abstracted out of the user's application.

IV. FRAMEWORK

In this section we provide a brief overview of the framework, which is divided into three parts, as illustrated in Figure 2. The Application Layer is discussed in IV-A, LocationSychronization Layer in IV-B, and the Communication Layer in IV-C. Each layer of the framework is comprised of several interfaces, which are implemented in our framework. The Application Layer resides inside the programmer's application and is an abstraction that provides our implementation access to the user's application. The application never directly communicates with the Communication Layer, but utilizes the high level constructs of async, finish, and place in the LocationSychronization Layer to create parallelism, to support synchronization, and to define locality. The LocationSychronization Layer manages the spawning of asyncs and finish barriers at different logical places. The LocationSychronization Layer directly communicates with the Communication Layer and uses a PlaceMapping, provided by the programmer, to translate logical places to physical places that are need by the Communication Layer. The Communication Layer uses physical places, a representation of the node's name, or process's id to inform the messaging library which place to send a message to. The Communication Layer handles the calls to the backend communication library the user has chosen. In our implementation, we chose DUP and MPICH2, two very different communication libraries to illustrate the ease of switching communication libraries.

A. Application Layer

The Application Layer is the entry point into the user's own application for our implementation and interface. In this section, we explain how the Application Layer interacts with the rest the LocationSychronization Layer and discuss the concept of logical and physical places. In Figure 9, we present the Application Layer, the layer that a developer or

```
interface Application_Layer {
public Application_Layer
      (LocationSychronization AFP)
  public void _pack_ASYNC(_async child)
  public void initialize_constants()
  public void Run_Main()
  public void thread_run(uint32_t method,
    BinaryBuffer params)
}
```

Fig. 9. Application Layer Interface

```
interface PlaceMap {
  private HashTable LogicalToPhysical
       <Place_T,PhysicalPlace>;
  public PhysicalPlace
  translateLogicalToPhyiscal(Place_t P);
}
```

Fig. 10. PlaceMap Interface

compiler writer would be required to implement to provide our implementation access to the user's application. `Run_Main`, a method we require in the user's application, provides a hook to start the application. Initilize_constants initializes static final global variables at each physical place because each physical place has its own address space. The developer or compiler writer must also implement or generate thread_run, a function that maps an async's method number to a method that contains the body of the async to be executed. Data needed by the body of an async should be transmitted with the async message and then passed along to the thread_run method. As seen in Figure 8 line 9, thread_run would call the method that maps to `ASYNC0` with data needed for execution of the async's body stored in params, line 11. Supporting the nesting of arrays and classes requires serialization, which is done in the `_pack_ASYNC` method in the Application Layer. Our retargetable compiler automatically generates the thread_run, `initialize_constants`, `Run_Main` and `_pack_ASYNC` methods. Our retargetable compiler generates the serialization methods for our implementation by analyzing each class and array to see if it is used within an async and by checking its definition to see if it includes a class or array. If both conditions hold, a serialization method is made to serialize the array or class and the nested array or class. This serialization technique allows for nested arrays and classes to be communicated across nodes.

The user's application does not directly communicate with the Communication Layer, illustrated in Figure 1, but communicates indirectly through public members of LocationSychronization Layer, explained in section IV-B. This separation allows for the user's application to operate on logical places instead of physical places. The user's application can acquire logical place information from the public place functions in the LocationSychronization Layer. For example `here()` will provide an instance of the user's application with its logical place. This abstraction restricts logical place information to the application and physical place information to the Communication Layer.

```
interface LocationSychronization {
  private class Task {
    uint32_t up_count;
    uint32_t down_count;
    pthread_mutex_t down_mutex;
    pthread_cond_t down_cv;
    Task parent;
  }
  public LocationSychronization
      (Communication C, PlaceMap P){
    public static void main();
    //finish constructs
    public void task_start_finish();
    public void task_end_finish();
   //async constructs
   public uint32_t task_dispatch(Async a,Place_T p);
   //place
   public Place_T here();
   public Place_T max_places();
   public uint32_t isfirst(Place_T p);
   public uint32_t islast(Place_T p);
   public Place_T place_first();
   public Place_T place_last();
   public Place_T place_next(Place_T p);
   public Place_T place_prev(Place_T p);
   public Place_T toplace(uint32_t p);
   //creates the thread for Run_Main
   private void _create_main_thread();
   // terminates all current running dispatchers
   private uint32_t dispatcher_terminate(Place_T p);
  //initializes logical to physical place data
  private uint32_t _place_init()
  private uint32_t dispatcher();
  }
  //logical place representation
    class Place_T {
    uint32_t place;
  }
  class PhysicalPlace{
  uint32_t place;
  }
  class Async {
    uint32_t method;
    uint64_t size;
    BinaryBuffer params;
    }
}
```

Fig. 11. LocationSychronization Interface

B. LocationSychronization Layer

The LocationSychronization Layer interface, illustrated in Figure 11, is where parallelism, synchronization, and the concept of places are abstracted. In this section, we explain the interface and how the high level X10 constructs async, finish, and place are represented in our interface. In our implementation, machines and processes are mapped to physical places while the X10 concept of places is mapped to logical places. This provides flexibility to the application, allowing for different hardware configurations, number of machines, and number of processors to be determined by a logical to physical place mapping without changing any source code. The programmer will provide a PlaceMapping via a file or program arguments, which maps places to node, CPUs, processors or

machines. This allows the application to run on a different PlaceMapping without modifying any source code and, if running under the same architecture, without recompiling any code. This also provides the application an efficient way to automatically scale, since the application can be designed with an abstract view of places. Further data in the same logical place should be accessible via shared memory while two logical places on the same machine should have separate address spaces. The PlaceMapping, illustrated in Figure 10, is a mapping of logical places to physical places, enabling the LocationSychronization Layer to pass logical places to the Application Layer and physical places to the Communication Layer.

Each physical place has its own address space and dispatcher, which manages incoming messages and directs them. The dispatcher is place-independent and handles all initialization of the library code and global static final variables in the application code. All messages come through a dispatcher, which handles the message accordingly. If the message is an async, it designates it for execution. The dispatcher also handles acknowledgment messages that are used to signal a parent async or finish of their child's completion.

Finish blocks, essentially barriers for all nested asyncs, are translated to a `task_start_finish` at the beginning of the block and a `task_end_finish` at the end of the block illustrated in Figure 8 lines 4 and 14. The `task_start_finish` function creates a Task object that is attached to any asyncs created within the block. This is so that the asyncs will be able to notify their parent that they have finish execution. The `task_end_finish` blocks until all child asyncs and their children recursively have completed execution. We support nesting in our runtime of async and finish by utilizing a method very similar to Misra and Chandy's [2] work on the Dijkstra-Scholten algorithm for termination detection [13]. In our scheme, all children notify the parent async or finish of their completion and each parent async or finish waits for their children's completion. A parent async or finish keeps track of the number of children that is spawned within its block, and once all children have reported completion, the parent async of finish stops blocking.

An async, which initialized parallelism, is translated to a `task_dispatch`, which packs the Async object and takes a place as a parameter that represents where to execute, as seen in Figure 8 lines 7-13. The Async object also stores the data needed for the execution of the body of the async. `task_dispatch` creates a Task object for the current async and attaches the parent async or finish Task to it. The Task object stores its parent's information in order to notify the parent async or finish of its completion. `task_dispatch` calls the Communication Layer through the WriteMessage function to send the Async object to the place of execution. As an optimization, if the place specified has the same logical place for destination and origin, the Communication Layer can be skipped and the async executed at the place of origin. This optimization showed us speedups from 2x to 10x, and the best results were with benchmarks designed for a shared memory model. When executing with only one place, our implementation utilizes a multi-threaded shared memory model due to this one-place optimization that avoids the Communication Layer. Otherwise, if multiple places are present in the PlaceMapping,

```
1  interface Communication {
2  private class Message {
3    uint32_t placeFrom;
4    uint32_t placeTo;
5    uint32_t messageType;
6    uint64_t size;
7    Task parentTask;
8    Async async;
9    }
10   public uint32_t WriteMessage(
11     Place_T placeToWriteMessage,
12     uint32_t messageType,
13     uint64_t sizeOfTask,
14     Task parentTask,
15     uint64_t sizeOfAsync,
16     Async async);
17 public uint32_t ReadMessage(
18     Place_T placeMessageFrom,
19     uint32_t messageType,
20     Task parentTask,
21     Async async);
22 }
```

Fig. 12. Communication Layer Interface

then the application is executed as a distributed model. Once an async is received at place by a dispatcher, it is then created locally by the LocationSychronization Layer through the `_create_async` function, which will initiate the async execution.

Execution of user's code is initiated through `thread_run` function of the Application Layer. Once the async is finished executing, an `ack_async` message is sent back to the place of origin to notify any parent async or finish that their child async has finished execution. In our implementation, the parent async will not terminate until all children async terminate. Switching from a thread spawning pattern to a thread pool pattern for creating parallelism with the async construct required only changing fifteen lines of code in the LocationSychronization Layer and only recompiling this layer.

C. Communication Layer Interface

The Communication Layer, illustrated in Figure 12, is the bridge between the LocationSychronization Layer and the messaging library. In this section, we provide an overview of the Communication Layer and the communication libraries. We implement our messaging library with two very different communication libraries: DUP and MPI-2. The DUP system is a multi-stream based communication framework that utilizes TCP streams and pipes for communication and is designed around a data flow graph where nodes represent physical places in our PlaceMapping. We utilized MPICH2 and OpenMPI, two message passing implementation of MPI-2 that utilizes point to point communication and ranks to represent physical places [14]. Utilizing a new messaging library would require changing only the WriteMessage, ReadMessage and `_place_init` functions throughout the whole implementation. In our implementation, WriteMessage required four lines of code change, ReadMessage required one line of code change and `_place_init` required five lines of code change. This totals to ten lines of code change to switch from DUP to

	Execution Time (secs)	
	DUP 4 Nodes	MPI 4 Nodes
In-house benchmarks		
Plasma	325.13	262.77
MapReduce	22.54	22.50
Java Grande benchmarks		
Linpack	18.25	15.24
Sor	10.86	5.23
Montecarlo	1.22	1.26
Crypt	1,568.53	1,607.85
Raytracer	57.89	46.49
Series	90.93	98.99
Moldyn	54.26	45.08
Sparsemm	66.56	66.15
HPC Challenge benchmark		
Stream	3.60	4.43

Fig. 13. Benchmark execution times(mean on 10 runs)

MPICH2 and requiring only the Communication Layer and the LocationSychronization Layer to be recompiled. The user application and the Application Layer did not require recompilation because each layer is independent and no changes occurred in these layers.

The WriteMessage function marshals a Message object into a TPL [2] image and sends the image to a given physical place. The place parameter passed into the WriteMessage function is the physical place translated from a logical place provided by the LocationSychronization Layer. On the receiving end of a WriteMessage is a ReadMessage that is waiting for an incoming message. Once a message is read at a given place, the message should be demarshaled from its TPL image and placed into a Message object where the values are then further unpacked and passed back to the dispatcher. Our implementation assumes that all network communication is handled by the messaging library and the messaging library handles buffers appropriately. If buffering is not handled by the communication libraries, then this should be done in the WriteMessage or ReadMessage functions. All of our implementations with DUP, OpenMPI, and MPICH2 required no additional buffering in the WriteMessage or ReadMessage functions.

V. EXPERIMENTAL RESULTS

We used 11 benchmarks, all written in X10-1.5 and translated to C. Eight are from the Java Grande benchmark suite [15]. One is from the HPC challenge benchmarks, and two were written internally. We chose these diverse benchmarks because they represent a wide range of common implementation issues. Our most complex benchmark is a plasma simulation program [16], [17], which we translated from Fortran with MPI to X10-1.5. The plasma simulation utilizes multidimensional distributed arrays over non-regular regions and calculates the vector force interactions of a large number of particles in a 2D space. Our MapReduce benchmarks sums a distributed integer array in typical map and reduce fashion.

[2]http://tpl.sourceforge.net/index.html

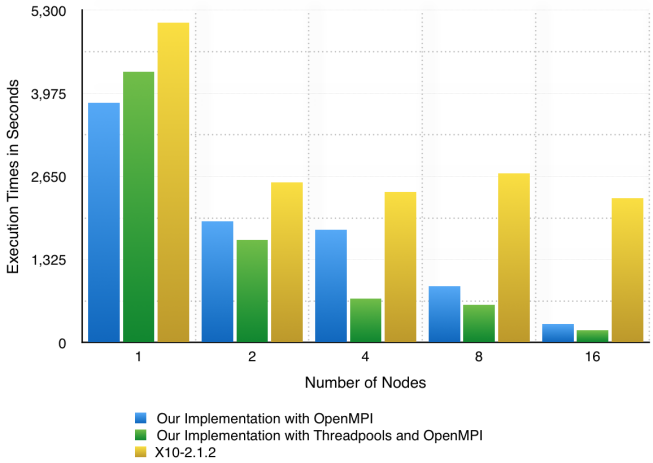

Fig. 14. Performance for series

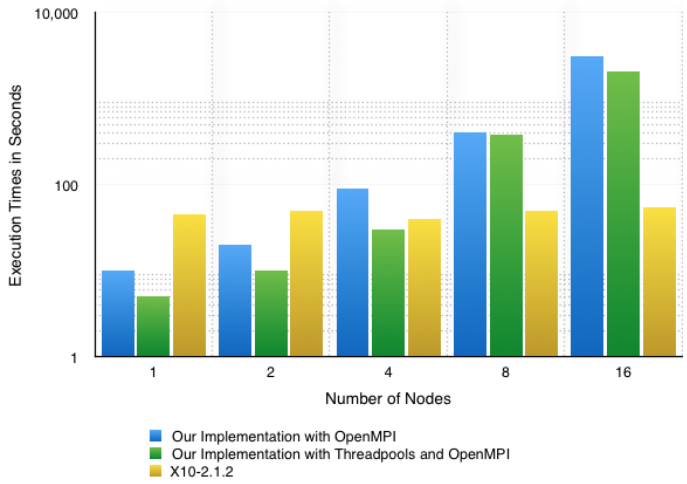

Fig. 15. Performance for stream

A. *Measurements*

We ran our experiments, in Figure 13, on a system with one quad-core Intel Xeon CPU 2.66GHz with 6 MB of cache and 32GB of RAM running the Ubuntu OS 2.6.32-28-server, and three iMacs with 2.33GHz duo core processors with 6 MB of cache and 2GB of RAM via 100Mbit Ethernet running Mac OS X 10.5.8. All four machines were connected via Ethernet connections. The benchmarks and the implementation were compiled for a 64-bit architecture but many of our benchmarks that utilize a small addresses space were able to execute for a 32-bit architecture. We ran our experiments, in Figure 14 and Figure 15, on the Hoffman2 cluster, with each node having an Intel Nehalem 2.53GHz CPU with 4GB of memory and all interconnected via Ethernet.

In our implementation, all benchmarks were compiled from X10 to C, in which the C source was compiled with gcc for execution. In X10-2.1.2, the source X10 code is compiled to C++ and later compiled with g++ for execution. We compared our implementation, using OpenMPI, to the X10-2.1.2 implementation, using sockets. No recompilation was done to either

implementation while scaling between 1, 2, 4, 8 and 16 nodes. We believe this allows for a fair comparison between IBM's X10-2.1.2 implementation and our own.

While comparing MPICH2 to DUP with the thread spawning pattern, the master node was the Intel Xeon machine while the three children nodes were iMacs. Figure 13 illustrates the average execution time of ten runs of our implementation on two separate communication frameworks DUP and MPICH2. The input sizes in Figure 13 for the Series and Stream benchmarks are about a hundred times smaller than the input sizes for comparing X10-2.1.2 to our implementation. This is because DUP is an experimental messaging library that cannot currently handle messages that are too large. Figure 14 and 15 illustrate the average execution time on the Hoffman2 cluster over five runs of the X10-2.1.2 implementation and our implementation. Series and Stream are the only benchmarks compared because the rest of the benchmark suite required arbitrary distributions, could not compile correctly, or produced run time errors on the IBM X10-2.1.2 implementation. All these benchmarks compiled and executed correctly on IBM's X10-1.5 and our implementation.

B. Evaluation

As illustrated in Figure 13, SOR, Plasma, and Moldyn performed better under MPICH2, while Crypt and Series showed better performance under DUP. These results illustrate the benefits of executing on different communication libraries. By switching communication libraries, performance gains were made without modifying or recompiling the programmer's application. In Figure 14, both our implementations outperform the X10-2.1.2 implementation on all numbers of nodes. Our thread pool implementation illustrates a speedup of about 6.2x when utilizing sixteen nodes and a speedup of about 4.5x when utilizing eight cores. Our thread pool implementation outperforms our thread spawning implementation, in Figure 14 and Figure 15, because the Stream and Series benchmarks have many asyncs with short execution cycles, favoring the thread pool pattern. Even though our implementation is not optimized for performance, it still outperforms the X10-2.1.2 implementation on Series, which has a distributed workload. This is likely because early versions of X10 were designed for a shared memory model and not a distributed model.

In Figure 14, we see that our thread pool implementation outperforms the X10-2.1.2 implementation while utilizing one node with a speedup of about 32.7x and two nodes with a speed up of about 6.3. While executing on one node, our implementation has an optimization to avoid the communication network utilizing a multi-threaded shared memory model. However, as we scale past two nodes, our implementation slows as it starts utilizing the communication network while the X10 implementation maintains a steady execution time. This is because the Stream benchmark was designed for a shared memory model, which is why our implementation outperforms the X10 implementation for 1, 2 and 4 nodes. The X10 implementation only outperforms our implementation as the distributed workload is overwhelmed by the costs of communication. The X10 implementation does not incur the cost of communication as the number of nodes increases because it does not distribute any work to the added nodes, avoiding the cost of communication all together.

VI. Related Work

Bikshandi et al. [7], to our knowledge, is the only other work that illustrates execution of active messaging in multiple places. A subset of X10 called FLAT X10 is demonstrated with a compiler that translates the source application written in FLAT X10 into C++ SPMD programs. A runtime system is exhibited and utilizes an active messaging infrastructure for communication based on the IBMs LAPI Low Level API. No support is given for nested async of finish blocks. Additionally, no support is given for different architectures or support for separate compilation. This work was recently extended with the X10-2.1.2 implementation, which now has support for nesting of async and support for communication frameworks beyond the LAPI API, such as sockets. Recompilation is required when switching communication frameworks and the implementation is strictly tied to the X10 language. The implementation can only use communication frameworks that match the templates they provided, such as LAPI and TCP, which languages like DUP do not fit. Another limitation of this work is it does not allow users to choose which nodes to map to their corresponding places, and requires the architecture and hardware of all nodes to be the same. This work also does not allow for the union of distributions the way in, which formed arbitrary distributions. This limits the problem scope that this solution can be used for, since arbitrary distributions are a requirement for benchmarks such as plasma. This work also does not allow for the nesting of value arrays and classes, a major inconvenience to the developer. Our implementation allows for arbitrary distributions, nesting of value arrays, value classes and out performs the X10-2.1.2 implementation in the majority of our results. Further although this work was intended for distributed execution it performs better on shared memory applications over distributed applications.

Given an algorithmic description, Paalvast et al. [2], [3] illustrate a method of automatic code generation for a distributed system. They present difficulties placed on the programmer when communication and parallelization are integrated into an algorithm, demonstrating it is hard to debug, and even minute changes can require major program restructuring. Flexibility allows for the communication schemes and synchronization to match the necessary problem. The source language Booster, a high level parallel programming language, is translated to distributed SPMD code in which communication and synchronization are generated automatically. The annotation language that describes the algorithm used in the generation of SPMD programs is not limited to Booster. No benchmarks of the system are demonstrated in either work.

Callahan et al. [18] offer an automatically generated message passing SPMD-like program from a sequentially shared memory application with the help of annotations that illustrate how data in arrays should be distributed. They suggest the annotations of data dependencies provide for portability, but expressed challenges with irregular or complex distributions of arrays. Data movement is tightly bound to the application since load and store statements are inserted into the translated source for data transfer. They suggest portability is important, and with a more flexible communication structure like our implementation, would improve the work. No implementation or performance numbers are presented.

Hiranandani et al. [4] present the parallel language Fortran

D, which enhances Fortran with data distribution. This work stresses the need for flexibility, code longevity, and retargeting to encourage the use of parallel computing in the scientific community. The Fortran D compiler outputs a SPMD program with explicit message passing. The compiler integrates communication directly into the code and is heavily reliant on owner computes rule, which states the process that holds the left-hand side element of an expression will perform the calculations. Fortran D is limited because of its reliance on the owner compute rule, which can lead to inefficient computation due to excessive communication and because it is limited to vector style computation. Benkner et al.[19] provide an overview of High Performance Fortran (HPF), a high-level data parallel language that is compiled to parallel Fortran with message passing for distributed systems. The compiler utilizes HPF mappings to determine data distributions. HPF does not support task parallel applications.

Cytron et al. [20] offer automatic translation of programs written in fork-join style and automatically introduce SPMD regions to express parallelism. Two limiting assumptions are made that there are no data dependencies between threads and that execution of parallel loops is deterministic, both of which we do not assume. Further this work is limited to the fork-join style, in which many benchmarks in our suite cannot be written.

Amarasinghe et al. [21] offer code generation techniques for optimizing communication and producing a SPMD program for each node, given a description of partitioned computation. The description aids in data-flow analysis used to generate the automatically parallelized code. Like our implementation, communication is automatically generated. However, unlike our implementation, the communication code is directly integrated into the code and does not allow for the flexibility of switching communication libraries. Retargeting the pattern of parallelism or communication would require recompiling the application.

Lapadula et al. [22] present a communication framework for C* code translated into C code. This framework is for mesh-connected MIMD multicomputers. The communication framework is integrated tightly into the generated C code. This integration of the communication code into the C code requires recompiling the application to retarget the communication.

Barton et al. [23] express the implementation of Unified Parallel C (UPC) runtime RTS and the compiler for UPC. UPC is a partitioned global address space (PGAS) language[24] similar to X10 and is also compiled to C as the runtime language. Static analysis is utilized to insert communication code into the runtime language. This insertion of communication code directly in the code would require rewriting the compiler and recompiling the user's code to retarget the communication, two disadvantages our implementation overcomes. Further inefficiencies result from the Shared Variable Directory (SVD) that UPC requires to be maintained across all places, putting a heavy burden on the communication backend. This is not required in the X10 languages representation of remote references aiding our implementation.

Darema et al. [25] present the SPMD shared memory model for Fortran using a Fetch-and-Add mechanism for enforcing synchronization. Techniques for debugging and SPMD programs are presented without any automation or compiler assistance. Synchronization code is directly inserted into the application requiring recompilation of the application if a change is needed. The computational model assumes a shared memory organization.

VII. LIMITATIONS

The design and implementation have controlled limitations. The runtime implementation does not yet support garbage collection for the Application Layer while the Communications Layer and LocationSychronization Layer memory is managed. We plan to optimize our implementation through message batching and statically determining which asyncs to run locally to prevent the communication from outweighing the benefit of distributed computation as in the X10-2.1.2 implementation.

VIII. FUTURE WORK

We hope to implement both a distributed garbage collector and a per-place garbage collector which we have prototypes for. We also hope to reduce the memory footprint of programs through static analysis to reduce the need for a garbage collector and to improve efficiency by reducing the number and size of messages. We also are aiming to demonstrate the support for even more heterogeneity by including GPUs in our cluster and having different execution engines with different source languages at each node. We are also aiming to develop automatically generated constraints during compilation to produce a valid mapping via executions on smaller sample sets, runtime load balancing and through code analysis.

IX. CONCLUSION

In this paper, we have presented a retargetable communication framework that supports scalability and can run on a variety of architectures and languages. Our three-layer framework offers users a natural way of separating the parallelism and communication library from the application. This separation allows for the communication library and parallelism to be changed without recompiling the application. Our implementations on both DUP and MPICH2 illustrate the minimal changes required to switch communication libraries. Our results illustrate that our implementation outperforms the X10-2.1.2 implementation and that our implementation can both scale without recompilation and provide portability. Our customization of serialization for the source application allows us to maintain performance while offering wealth of features. Our main goal for the future is to optimize our implementation to avoid communication when the workload is too small and to further illustrate the flexibility of the framework.

REFERENCES

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," Dr. Dobbs journal, vol. 30, no. 3, pp. 202–210, 2005.

[2] J. Misra and K. M. Chandy, "Termination detection of diffusing computations in communicating sequential processes," ACM Trans. Program. Lang. Syst., vol. 4, no. 1, pp. 37–43, Jan. 1982. [Online]. Available: http://doi.acm.org/10.1145/357153.357156

[3] MPI Forum, "Message Passing Interface (MPI) Forum Home Page," June. [Online]. Available: http://www.mpi-forum.org/

[4] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling fortran d for mimd distributed-memory machines," Commun. ACM, vol. 35, no. 8, pp. 66–80, Aug. 1992. [Online]. Available: http://doi.acm.org/10.1145/135226.135230

[5] K. C. Bader, T. Eißler, N. Evans, C. GauthierDickey, C. Grothoff, K. Grothoff, J. Keene, H. Meier, C. Ritzdorf, and M. J. Rutherford, "Distributed stream processing with dup," in Network and Parallel Computing. Springer, 2010, pp. 232–246.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," Acm Sigplan Notices, vol. 40, no. 10, pp. 519–538, 2005.

[7] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen, "Efficient, portable implementation of asynchronous multi-place programs," in ACM Sigplan Notices, vol. 44, no. 4. ACM, 2009, pp. 271–282.

[8] N. T. Karonis, B. Toonen, and I. Foster, "Mpich-g2: A grid-enabled implementation of the message passing interface," Journal of Parallel and Distributed Computing, vol. 63, no. 5, pp. 551–563, 2003.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine et al., "Open mpi: Goals, concept, and design of a next generation mpi implementation," in Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer, 2004, pp. 97–104.

[10] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, Active messages: a mechanism for integrated communication and computation. ACM, 1992, vol. 20, no. 2.

[11] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick, "Titanium language reference manual, version 2.19," UC Berkeley Tech Rep. UCB/EECS-2005-15, Tech. Rep., 2005.

[12] U. Consortium et al., "Upc language specifications v1. 2," Lawrence Berkeley National Laboratory, 2005.

[13] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," Information Processing Letters, vol. 11, no. 1, pp. 1–4, 1980.

[14] N. S. Evans, C. GauthierDickey, C. Grothoff, K. Grothoff, J. Keene, and M. J. Rutherford, "Simplifying parallel and distributed simulation with the dup system," in Proceedings of the 2010 Spring Simulation Multiconference. Society for Computer Simulation International, 2010, p. 154.

[15] L. A. Smith, J. M. Bull, and J. Obdrizalek, "A parallel java grande benchmark suite," in Supercomputing, ACM/IEEE 2001 Conference. IEEE, 2001, pp. 6–6.

[16] D. Dauger, V. Decyk, and J. Dawson, "Using semiclassical trajectories for the time-evolution of interacting quantum-mechanical systems," Journal of Computational Physics, vol. 209, no. 2, pp. 559–581, 2005.

[17] J. Tonge, D. E. Dauger, and V. K. Decyk, "Two-dimensional semiclassical particle-in-cell code for simulation of quantum plasmas," Computer physics communications, vol. 164, no. 1, pp. 279–285, 2004.

[18] D. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors," The Journal of Supercomputing, vol. 2, no. 2, pp. 151–169, 1988.

[19] S. Benkner and H. Zima, "Compiling high performance fortran for distributed-memory architectures," Parallel Computing, vol. 25, no. 13, pp. 1785–1825, 1999.

[20] R. Cytron, J. Lipkis, and E. Schonberg, "A compiler-assisted approach to spmd execution," in Proceedings of the 1990 ACM/IEEE conference on Supercomputing. IEEE Computer Society Press, 1990, pp. 398–406.

[21] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in ACM SIGPLAN Notices, vol. 28, no. 6. ACM, 1993, pp. 126–138.

[22] A. Lapadula and K. P. Herold, A retargetable C* compiler and run-time library for mesh-connected MIMD multicomputers. Citeseer, 1992.

[23] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral, "Shared memory programming for large scale machines," in ACM SIGPLAN Notices, vol. 41, no. 6. ACM, 2006, pp. 108–117.

[24] D. Bonachea, P. Hargrove, M. Welcome, and K. Yelick, "Porting gasnet to portals: Partitioned global address space (pgas) language support for the cray xt," Cray User Group (CUG09), 2009.

[25] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for epex/fortran," Parallel Computing, vol. 7, no. 1, pp. 11–24, 1988.