High-level Design Document: The X10lib Design v0.91

Vijay Saraswat, IBM TJ Watson Research Center
Sriram Krishnamoorthy, Ohio State University
Ganesh Bikshandi, IBM India Software Lab
Rajkishore Barik, IBM India Research Lab

March 15, 2007

Abstract

X10lib is a runtime system for X10, designed for tightly-coupled clusters of multiprocessors. It supports a partitioned global address space, multiple places, global datastructures (arrays), and dynamically spawned activities and synchronization operations between these activities. The library is intended for use by the X10 compiler as a runtime library, and also by programmers in C/C++ wishing to write code in the X10 style.

This document presents a snapshot of the high-level design of X10lib. A companion note will document the actual APIs exported by X10lib. This document will be kept uptodate on the X10 website, `x10.sf.net`.

1 Introduction

1.1 Design Goals

X10lib is a runtime system for X10 [3]. We list the following explicit design goals for X10lib:

- X10lib should permit current C/C++ MPI programmers to program with X10 concepts such as global shared memory, multi-threaded places, asyncs, futures, clocks and one-sided memory operations.
- X10lib should export an interface to the compiler similar in spirit to the interface provided by the Java Virtual Machine. The interface must provide specialized entry points for common cases (e.g. allocating 1d, 2d, 3d rectangular arrays).
- It should be possible to spawn activities in the current process or on remote processes, and detect quiescence and termination of these activities.
- The library should be available in C and C++ and usable within MPI programs.
- The library should be layered on top of a *Messaging API*. The Messaging API should be efficiently implementable on top of LAPI [1], ARMCI [4], GASnet [2] and other similar high-peformance messaging systems.

  We may choose to build directly on top of GASNet.
- Data-structures should be designed for scalability.
- The design should efficiently execute programs that are communication-intensive and programs that are compute-intensive. It should permit the programmer to specify how many processes to run on each node of the cluster, how many places to run on each process, and how many threads to assign to each place.

Initially bindings will be provided for C and C++. Bindings for Fortran and Java are also planned.

The following are explicitly *not* design goals for X10lib:

- X10lib is a C/C++ API: it will not offer any explicit support for statically proving properties such as deadlock-freedom. (Such properties are provided at the language level for X10).
- This version of X10lib will not provide any support for garbage collection. The user is responsible for automatically managing the lifetime of objects. (Functions will be provided to allocate space in the global shared memory.)
- This version of X10lib will not support the dynamic creation of places.
- This version of X10lib will not support checkpointing of places. It will not support migration of places from one process to another.
- Distributed data strutures other than arrays will be provided in a subsequent release.

1.2 Summary of desired functionality

X10lib provides functions to do the following:

- Get a remote (globally valid) reference for a local data structure.

  Such references can be passed from place to place and still remain valid. Two such references may be checked locally (without any communication) to determine if they point to the same data structure.
- Spawn an async at a specified place, with a given function pointer and arguments, and clocked on given clock set. (The place may be specified either through a specific structure naming the place or through a remote reference.)
- Create a clock, and perform clock operations (next, resume, drop).
- Spawn a future at a given place, with a given function pointer and arguments, and clocked on given clock set, and return the future.
- Force the future. (This causes the current thread to block until the future value has been computed, and must therefore be performed at the top-level.)
- Spawn activities in parallel at a given vector of places.
- Terminate the current activity. (This does not terminate the current pthread, merely causes it to look for another activity to execute.)
- Call finish to suspend on the termination of all asynchronous activities in the scope of the finish.
- Perform an atomic operation with a given piece of code.
- Throw an exception (to be caught by an enclosing finish).
- Initialize a global data structure by allocating local arrays at given places.
- Perform operations on the local portion of an array – read/write/atomically update/iterate over elements in storage order/region order.
- Copy a portion of an array from the local place to a remote place, or from a remote place into the local place.
- Perform reduction operations on a global array.
- Free global or local data structure.

Additional intrinsics capturing common idioms for communication that can be efficiently executed on modern hardware will also be provided.

Target architecture The intended target architecture for the initial X10lib implementation is a cluster of SMPs connected through a high performance switch. For instance, a cluster of 64-way Power5 SMPs connected by a Federation switch. We view this as the best current approximation to the PERCS machine being designed for 2010. We expect the current design to be applicable to the PERCS machine though the details may change significantly.

1.3 Overview of the design

An X10 *job* consists of a number of X10 operating system *processes*, each running on an SMP. (A single SMP may host several processes in the same job.) Each process *hosts* one place, and uses one or more application-level threads to execute the asyncs in these places. Each process uses the messaging layer to communicate with other processes. On being launched a job consults some configuration information to determine how many processes are to be started at different nodes in the cluster. On being launched each process initializes its internal data structures, and synchronizes with all other processes to coordinate its initial state.

Each process maintains a collection of threads and a (double-ended) queue of asyncs per place. The threads associated with a place suspend waiting for a new activity to be added to its queue. This activity may be added as a result of the execution of an async at a different place in the current process, or on receipt of an "active message" from the messaging layer.

When a thread is awakened it pulls the next async out of the queue, determines whether the blocking condition (if any) associated with the activity is satisfied and then executes it to completion. If the condition is not satisfied then object corresponding to the async is suspended on the condition variable. It will be awakened (and added again to the activity pool) once there is reason to believe its condition has changed.

An async is represented as an object with a particular interface. Asyncs are classified statically based on the characteristics of the code they run: *nonblocking* (code does not perform a conditional wait on some variable), *local* (executing in the same place), *immediate* (can be executed without acquiring a lock). Blocking asyncs are required to be *top level* blocking: the call to conditional wait must not occur within a method call. During execution an activity A may spawn new asyncs. If the new async is to be executed at a different place in the current node and is immediate, it may be performed immediately by the thread executing A. If the new async is local and nonblocking (or immediate) it will be executed immediately (by the spawning activity). For all other asyncs intended to execute at another place in the same process, an object is created and added to its queue of asyncs. If the new async is intended to execute at a remote place then a LAPI active message is queued for the target place.

The compiler may use static analysis and type information to determine that an activity should be inlined (e.g. because it is local and immediate). Such activities do not even show up as activities in the runtime system. In particular the compiler is strongly encouraged to examine the bodies of asyncs and determine if they can be classified as immediate/local/nonblocking.

An async may be associated with some synchronization-related data-structures, such as the clocks on which it is registered.

Implementation of atomic operations X10lib provides special kind of objects, *monitored* objects (e.g. futures). Each monitored object is a normal X10 object together with a data-structure (the suspension queue) for recording the activities that are suspended on this datum, waiting for it to change state. Writes to fields of monitored objects result in the suspension queue being examined. The suspension predicate for each async in the queue is evaluated, if it is true the async is moved from the suspension queue to the async queue associated with the place.

¡

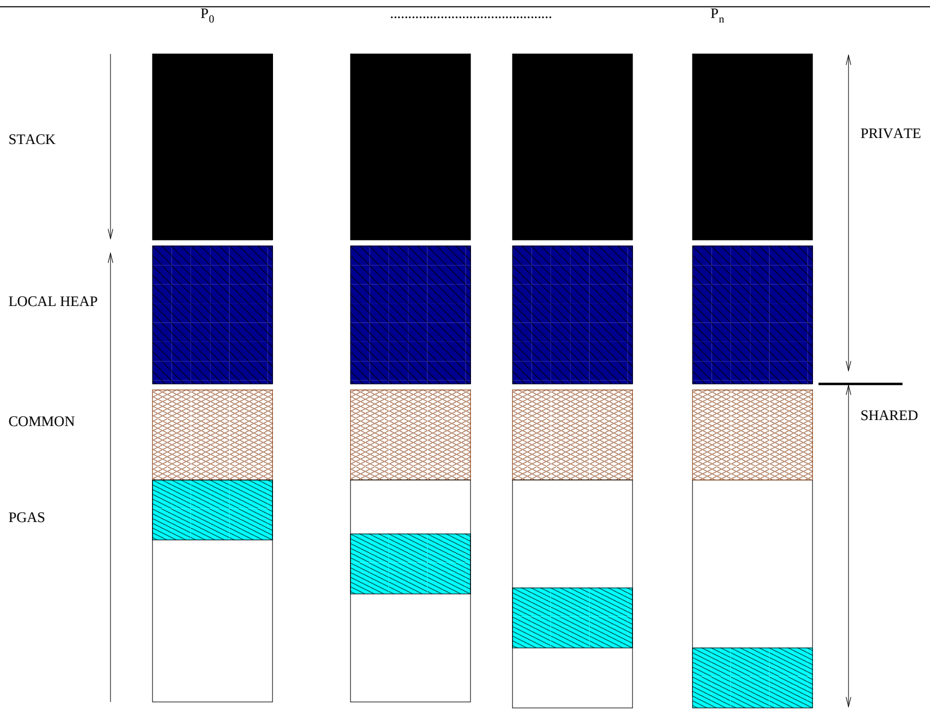

Figure 1: X10 The address space of an X10lib process

2 Design overview

2.1 Global Address Space

The global address space (GAS) view is essential to implement any PGAS language. X10lib will provide this abstraction. Before proceeding to the details, let us define certain terminologies. An address space (AS) is k-bits if it can support addresses from 0 to $2^k - 1$. We will be concerned with two address spaces – the Local Address Space (LAS), representing the address space in a single process running on an SMP, and the Global Address Space (GAS), representing the entire address space available to the computation (= collection of processes, scattered on multiple SMPs). The LAS is divided in to two regions – local and shared.

The shared region of a process is divided into two regions – *common* and *hosted*. The higher n-bits of the 64-bit address (i.e. a_{63-x} to a_{63-x-n}) are used to address the common section. Each process will store its private data in the common section. For example, the meta-data (region, distribution, base address) of a distributed array and the *single* variables [5] are stored in the common section. So each processor will need to provide backing store (through its virtual memory system) for this portion of the address space. When the processor wishes to access the contents of a location in this region of the address space — it merely reads the location in its local address space.

Now we divide up the remaining address space among the 2^p nodes by using the next p higher-order bits (ie. $a_{63-x-n-1}$ to $a_{63-x-n-p}$) to identify the node. The portion of the address space that is allocated to a given processor P is said to be "hosted" at P. We call the union of the address spaces hosted at each processor the "partitioned address space." Thus the total shared address space is divided up into the partitioned address space and the common address space. The entire addres space is pictorially represented in Figure 1.

Whenever a processor P desires to read or write an address A in the partitioned address space, it first determines the processor Q at which this address is hosted, and then asks Q to read or write the address on its behalf (using LAPI remote gets and puts). Note that A is valid in P's address space but will never be referenced. So P's virtual memory tables will need to allocate real memory only to the portion of the address space that is hosted at P or is common.

An advantage of such a representation is that address arithmetic can be performed locally. Suppose that a processor wishes to get the contents of $A[i]$. The code will read the metadata for A from locations in the Common Address Space. Using this information (e.g. information that says the array is block distributed, and provides the block size), the code will compute the target address for $A[i]$. Note that the code sequence necessary to do this calculation is *identical* across all nodes. Now that the target address is known the code will determine whether it is local or remote by examining the higher order bits. If it is local then it reads the location from its local memory, otherwise it uses a LAPI-get to read the memory.

It is not required that the contents of memory in the common address space are actually identical across all processors. Often they will be. But sometimes they will be different. For instance, in the case of global array A, the meta-data will be stored in the common address space. The first (64-bit) word at this address will contain a pointer into the portion of the partitioned address space hosted at this node which contains the data for the local portion of the array. Subsequent words may contain the meta-data for the array – the contents of these words may be identical across all nodes.

The point is that an address in the Common Address Space "means" the same to all processors, e.g. the global array A represented by the address 0x000078ABCDDDDDD in the Common Address Space. So processor P can send a message to any other process Q with this address, and Q will be able to use it to get to its local data for A. The contents of the first word at this address will be *different* for each processor Q – but will mean "the same," i.e the local portion of the array.

2.1.1 Implementation

The new operator in C++ should be overloaded and thus transparently (i.e. without explicitly telling at an allocation site) use a custom memory allocator for objects of certain types. A sufficiently large chunk of the address space is allocated ahead of time using a special OS call (e.g. **mmap**) [2]. The distributed objects and arrays are placed in this large chunk as the program executes. The custom memory allocation for PGAS implementation (done in C++ or C) is useful for two reasons.

First, depending on the communication subsystem (e.g. LAPI) virtual memory pages that belong to the PGAS may have to 'pinned', i.e. marked to stay resident in main memory. This is necessary to serve remote accesses timely without the risk that a remote access hits into virtual memory that is paged out.

Secondly, one can allocate the chunks of a distributed array in different nodes at the same address offset. If both contiguous array variables are allocated at the same offset in the virtual address space in each node, then the computation of the virtual address of any array variable can be done easily at the source node without any address translation at the home node.

2.1.2 Remote References

In shared memory systems where all accesses to data are through global pointers, all pointers are valid at all places. In such a scenario, the programmer can separate the problems of data distribution and computation partitioning. This greatly simplifies programming. In a distributed memory machine the shared memory abstraction incurs a performance penalty. This can be reduced by automatically translating global pointers to local pointers where possible. In the context of X10, every pointer carries the place of residence in its type. The compiler can translate global reference to local references. Asyncs that are thus determined to operate on local pointers can be inlined.

The runtime does not provide any such support and requires the user to explicitly distinguish local and remote references. In X10lib, remote references encapsulate global pointers, potentially making them safer. In a typical one-sided library, global pointers are identified by a place id and a pointer at that place. These two components are visible at all places, potentially allowing dereference of a remote pointer at the current place, resulting in undefined behavior.

X10lib allows access to the pointer encapsulated by a global pointer, only if that global pointer is local, i.e., points to a data structure at the current place. This prevents accidental dereference of the global pointer

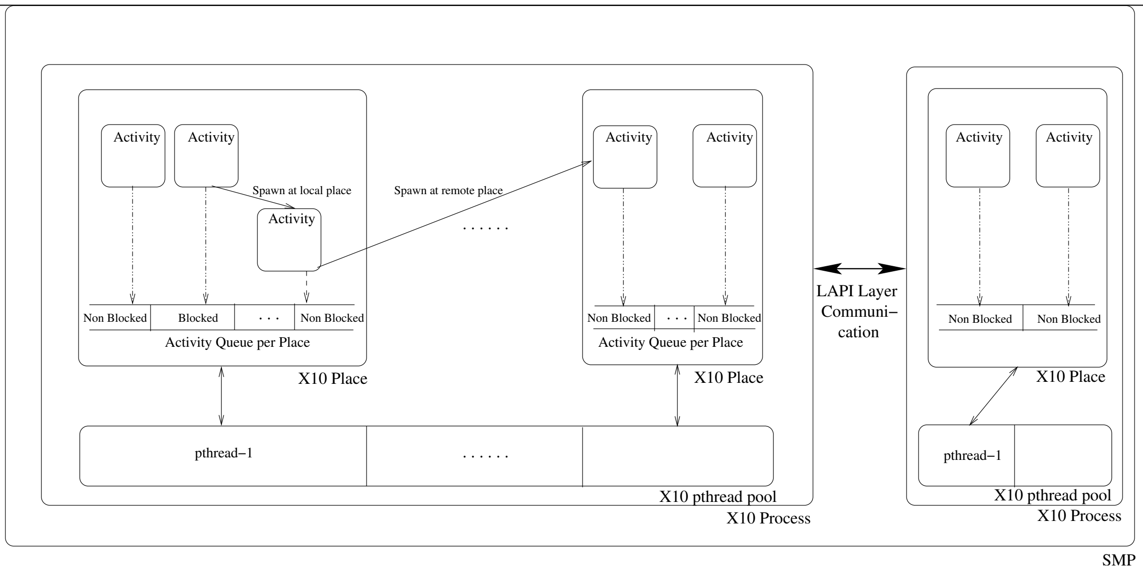

Figure 2: X10 deployment on an SMP node

at an incorrect place. To access an address that is not local (i.e. read or write), LAPI put/get methods should be used.

2.2 Deployment

A key feature in X10 programming model is the concept of *places* which establishes a mapping between a set of *activities* and a set of locations in the partitioned global address space(*PGAS*). The mapping of places to physical processors and memory is known as *deployment*. The *deployment* scenario plays a vital role in the design of efficient X10 execution environment. In this section, we will discuss on efficient X10 runtime design choices for deployment in a cluster of SMP nodes.

An X10 program written using multiple places can emulate multiple places within a physical SMP node. Various design choices can be thought of:

- A set of X10 *places* are deployed in a single SMP node. Each X10 place is assigned exactly one pthread for computation.
- A set of X10 *places* are deployed in a single SMP node. A set of X10 places are assigned one pthread for computation.

In the first iteration, we will be considering the first choice i.e., each X10 place is deployed on one pthread of an SMP node. Figure 2 is a pictorial view of the activities executing within a single SMP node. An X10 job consists of a set of X10 processes. The mapping from processes to processor cores in SMP is provided in the configuration file. Each X10 process hosts an X10 place. Each X10 place is associated with an "unbounded" activity queue which keeps track of both *blocked* and *non-blocked* activities. Blocked activities will be tagged and will not be selected for execution until they are untagged by the predicate of the data structure on which it is blocked. Non-blocking activities continue execution till completion. When there are activities in the activity queue, the X10 process pool executor will pick the top most activity from the queue and assign it to the designated pthread for execution. If the executing activity is non-blocking, it executes to completion. However, if the activity is blocking, it will be tagged as "blocked" on a predicate of a data structure and is put back in the queue when it is awakened by the same data structure. Activities of a X10 place are allowed to create both non-blocking and top-level blocking activities at remote places.

2.2.1 Activities

X10 programming model allows activities to be either place local or remote. These activities can execute any arbitrary code within its body including blocking operations. The blocking property of an activity is derived from the fact that it uses one of the following X10 constructs in : *finish*, *future-force*, *clock-next*, and *atomic-blocks*. In the first iteration, we will allow top-level blocking activities and non-blocking activities. Note that top-level blocking activities are those which does not have a predicated wait in method body. This information can be determined statically.

Local non-blocking activities may execute like an ordinary function call. The compiler may inline such activities. Local blocking activities are enqueued in the local activity queue for execution. During the course of execution if the activity blocks, it suspends its execution and waits on a predicate of a data structure to be set. As soon as the predicate of the data structure is set, the blocked activity is again enqueued and allowed to continue its execution. Remote blocking activities are enqueued at the destination place's activity queue and follow the same semantics of blocking as local blocking activities. Remote non-blocking activities are enqueued at the destination place's activity queue.

Operations on Activities:

- **Spawning new activities:** New non-blocking child activities are always allowed to be created. Similarly blocking child activities are allowed only if they do not have conditional wait in method bodies. Note that arbitrary blocking activities are not considered in the first iteration.
- **Executing blocking operation:** Mark the activity as blocked in the activity queue and allow other activities to progress.
- **Re-enabling blocked activities:** When the predicate of the data structure is set, the blocked activity is re-enabled in the activity queue and allowed to proceed with the execution.
- **Registration on clocks:** Activities during creation or execution can register with clocks. Clocks keep track of all activities that are registered on them at a given point of execution.
- **Deregistration on clocks:** During execution, activities can deregister themselves from certain clocks by explicit API calls. This will implicitly mean that these activities will not participate in subsequent clock quiescence.
- **Resume on clocks:** When an activity invokes *resume* operation on a clock, it intends to say that it has finished computation of the current phase and moves to the next phase.
- **Waiting for clock quiescence:** Activity invoking *next* operation needs to wait for global quiescence of all clocks that it is registered with.
- **Critical region code execution:** Codes protected under atomic sections are executed exclusively with respect to other activities within the context of a X10 place.
- **Propagate exceptions:** During execution an activity can throw exception which will be propagated to the innermost *finish* scope.
- **Collective operations:** A set of activities can participate in performing an operation collectively i.e., finding the sum of a distributed X10 array.

Example:

Consider the following code fragment from RandomAccess benchmark of HPC Challenge benchmark suite:

```
1: finish ateach (point p[i]: ranStarts.distribution) {
2:    long ran = nextRandom(ranStarts[i]);
3:    for (point count: [1:N_UPDATES_PER_PLACE]) {
```

```
4:       final int j = f(ran);
5:       final long OK = smallTable[g(ran)];
6:       async(table.distribution[j])
7:           atomic {
8:              table[j] = table[j] ^ OK;
9:           }
10:      ran = nextRandom(ran);
11:   }
12:}
```

The above piece of code updates the *table* elements based on a randomly computed index. There are two distributed X10 arrays: *ranStarts* and *table*. Elements of these arrays are distributed in the partitioned global address space and is shared across SMP nodes, X10 processes, X10 places and X10 activities. Statement 1 creates a number of asynchronous activities at various places based on the distribution of *ranStarts* points. Each of these activities subsequently update the *table* elements in Statements 6-9. The number of updates at each place is bounded by a constant *N_UPDATES_PER_PLACE* in Statement 3. Note that *smallTable* is not distributed and all of its elements are resident in a fixed predefined initial place. Since the updates to table elements are performed across various X10 places, Statement 8 is protected in *atomic* sections in Statement 7.

2.2.2 Atomics

Atomic sections are implemented using locks. Each place will have a set of locks. An activity may request to perform an atomic operation at a place which in turn will be translated to obtaining the locks at the designated place before performing the operation. These locks are globally ordered, precluding deadlocks. Obtained locks cannot be passed from one activity to another, and all locks required by an activity need to be obtained together.

On a cluster of Power5 SMP nodes, we primarily have two types of atomics: *primitive* atomics and *compound* atomics. *Primitive* atomics do not block within the same SMP node and use `lwarx` and `stwcx` instructions to atomically update within a SMP node. However, across SMP nodes, we need *compound* atomics which explicitly obtains lock at the designated place to perform the operation and are blocking in nature. It can be observed that primitive atomics designed specifically to improve performance on a cluster of Power5 SMP nodes.

Consider the RandomAccess code fragment given in Section 2.2.1. The spawn of asynchronous activities in Statement 6 can be optimized to leverage the *primitive* atomics concept on Power5 and can get rid of the cost of creating new asynchronous activities and obtaining locks within the same SMP node.

2.3 Arrays

X10 arrays provide support for the creation and manipulation of distributed multi-dimensional arrays. Once created, any activity can operate on arbitrary regions of the array, potentially through asyncs. The implementation of X10 arrays in X10lib will provide the interface for all X10 array operations viz., point-wise, reduction, scan. Also, an efficient iterator function will be provided. The maximum rank (i.e. the number of dimensions) of the array supported by the library will be fixed to a small constant (say 7).

A distributed array consists of two key components – meta-data and data. The meta-data consists of information about the array, like, shape (rank and size of each dimension), distribution etc. The meta-data is stored in the local heap of each process and is immutable. The data consists of the actual array data. The data is allocated in the hosted address space of each process. The meta-data and data are referred to as `array_info` structure.

We categorize the distributed arrays into three kinds – *regular*, *semi-regular* and *irregular*. The regular arrays are created only by the root process (ie. the process that executes the first activity) and have very regular distribution – that is, all the processes own a chunk of data and the chunks are of same size in all

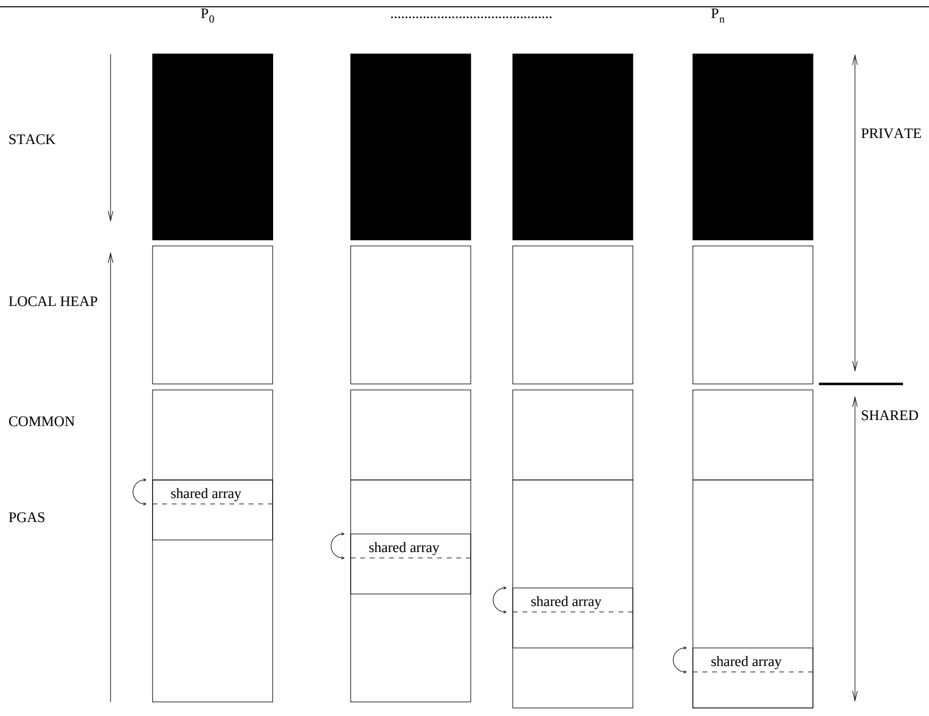

Figure 3: A sample layout of a *regular* distributed array

the nodes. Semi-regular arrays satisfy the property that it can be allocated by the root process; the other two properties of regular array do not hold. Irregular arrays are the most general arrays which don't have any of the above 3 constraints of the regular arrays. For efficiency in array access operations, we plan to allocate each of these arrays differently.

2.3.1 Regular Arrays

For regular arrays, we plan to exploit the benefits of GAS and the compiler. The chunks of regular arrays are allocated in a special region called *array* region. This array region is a part of the PGAS. All the process allocate its chunk in the same base address, with the first word containing a pointer to the meta-data. The rest of the words contain the data. This allocation scheme is shown in Figure 3.

The main advantage of this allocation is that an array access like $a[i]$ will be directly compiled to the following target code by the compiler : $get(((a.base\&mask)|node\_id) + local\_offset(i))$. Thus, no extra table lookup is required, unlike the allocation scheme that we will see in the following sections.

Consider again the RandomAccess code of Section 2.2.1. The code has two distributed arrays - `table` and `ranStarts`. Both these arrays are created in the beginning of the program as follows:

```
1: final long[.] table = new long[block(TABLE_SIZE)]
2:                (point p[i]) { return i; };
3: final long[.] ranStarts = new long[unique()]
4: (point p[i]) { return C.starts(N_UPDATES_PER_PLACE*i); };
```

The root process invokes the array constructor. The array constructor determines the base address where the array needs to be allocated. It then sends a message to all the processes to allocate the array at the same base address in their section of PGAS along with initial data and meta-data of the array. The processes receive the message, allocate the array and initialize it. The array construction terminates only after the message is received and the arrays are allocated and initialized at the receiving end.

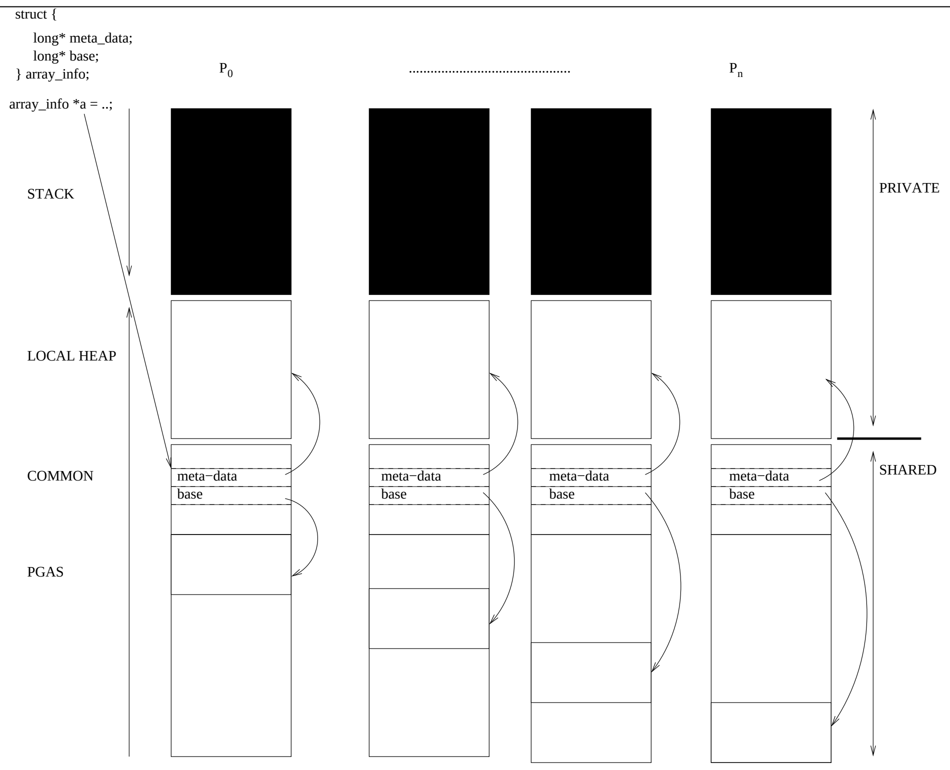

Figure 4: A sample layout of a *semi-regular* distributed array

2.3.2 Semi-regular Arrays

For semi-regular arrays, the `array_info` is stored in the common address space. The root process sends a message to all the process to allocate the `array_info` in the same address. Upon receiving this message, each processor allocates the space for `array_info` (in the common address space), meta-data (in the local heap) and data (in the hosted space). A sample layout of a distributed array in the address space is shown in the Figure 4.

For example, in the RandomAcess code listed in the previous section, the root process invokes the array construction. The root process sends a (LAPI) message to each process requesting a block of the array to be allocated on that process and initialized. Initialization information and other information like the distribution are sent to the processes in the message. Upon receiving the message, each process creates a meta-data for the array in its local heap, and allocates a region in its hosted address space (cf. Section 2.1) for the local data section of the array. It also stores the `array_info` in the address sent by the root process in the common address space. The call to the array creation routine returns only after the message is received by the processes involved, and the array sections are initialized in each process.

2.3.3 Irregular Arrays

Irregular arrays are the most general arrays. For these arrays, we generate a unique handle to represent the distributed array. This unique handle can be generated by the concatenation of process ID and a local counter. (ie. ⟨process id,local counter⟩). A hash-table is allocated in the local memory of each process. The unique handle is used to lookup in to the hash table and obtain the `array_info` for the given handle (array). This scheme is shown in the Figure 5.

For example, in the RandomAccess code, the activity that creates the arrays (not necessarily the root activity), constructs an unique handle using the above mentioned scheme. It then sends this handle and

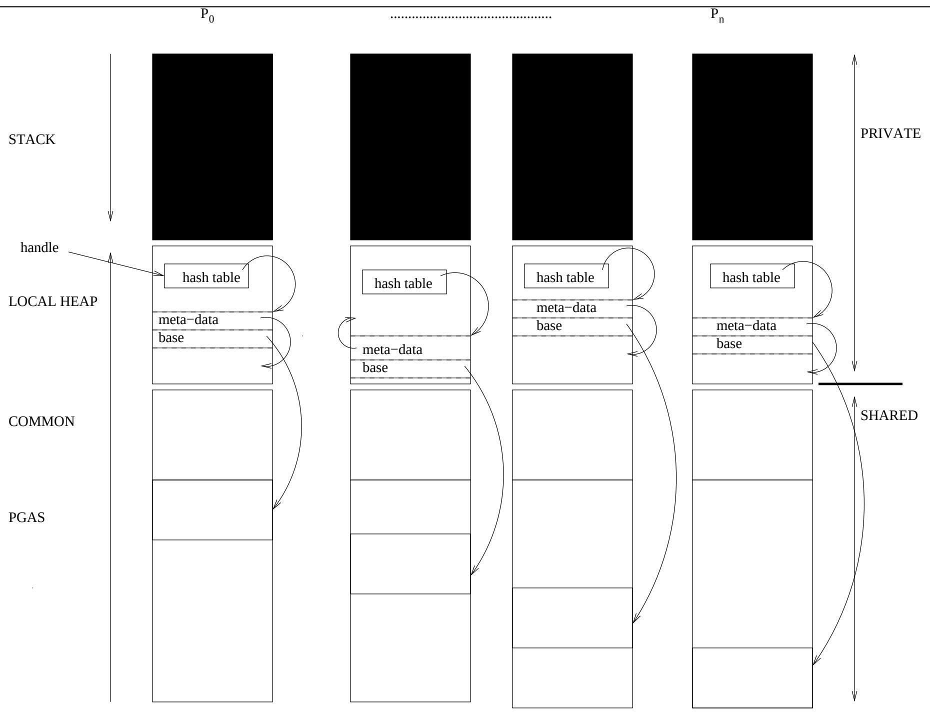

Figure 5: A sample layout of an *irregular* distributed array

other information (initial values, meta-data etc.) to the other processes that need to allocate a section of the distributed array. On the receiving end, each process allocates the `array_info` in its local memory. The data for the array is allocated in their section of PGAS. The pointer to the `array_info` is stored in the hash table entry corresponding to the unique handle sent by the activity that invoked the array constructor.

2.4 Execution Environment

There are two modes of using the X10lib. In the first mode, the compiler compiles the X10 source program and links it with the X10lib. The programmer issues a command to execute the resulting executable using a script. The script is responsible for sending the executable across the network to each of the SMP nodes. The script also reads a configuration file. The contents of the file will specify how many processes need to be created for this computation, the hosts on which these processes need to be run, the number of places to be created in each process, and the number of threads to be created for each place. The root activity will be run by a master thread. The other threads in each process will suspend waiting for incoming asyncs. By a thread, we refer to the application level threads, which may or may not be the same as a `pthread`.

X10lib is available for direct use by programmer too, following MPI style operation. The programmer is responsible for writing an SPMD-like code that uses the X10lib intrinsics. `X10::Initialize()` initializes the runtime system in each MPI process, and `X10::finalize()` performs any necessary clean-up. All invocations of X10 API must be after initialization and before clean-up. The X10 runtime object at the current place is implemented as a singleton, and is accessible by the `TheX10()` method. A place is identified by an integer. The number of places is fixed and does not change during the execution. Methods `TheX10::here()` and `TheX10::maxPlaces()` return the current place id and the number of places.

3 Conclusion

We have presented a design for X10lib, the runtime library for X10 programs, intended to run on a cluster of multiprocessors, running LAPI. Implementation of this design has begun.

A Background

A.1 X10

The X10 language provides support for a non-SPMD style functional-parallelism oriented programming. The unit of parallelism in an X10 program is an *activity*. The X10 computation starts with a single activity. Activities execute in *places*; a single X10 computation runs over many places, scattered over multiple nodes in a cluster. Activities are resident in the place in which they are created, and they can read and write data in the place in which they run. The may also create new local data items. Data items once created do not move. An activity can spawn other activities to be executed in parallel in the current place or in other places. All access to remote data is through activities and serialization of remote objects.

An activity may also create a global data-structure, such as a global array. Such a data-structure has state scattered across multiple places. Operations on the data-structure may be invoked by an activity in any place and typically result in computation across many places.

Of particular interest in X10 are distributed arrays. These are organized around a *region* – a data-structure representing the set of *points* over which the array is defined – and a *distribution*, a partitioning of the region over some subset of places. Sometimes the programmer may wish to operate upon an array in parallel using multiple activities within the same place; in this case a *tiled region* may be used to divide the array up between these activities. A tiled region is an array of regions. (In subsequent versions of the language we intend to introduce hierarchically tiled arrays.)

X10 supports a notion of *immutable* data. Objects that are immutable may may be freely copied from place to place by the implementation and are hence not associated with a particular place.

In addition to the activity-based communication model, X10 supports libraries for efficient copying of sections of arrays from one place to another.

Finally, X10 supports a few constructs for coordinating multiple activities. As noted above, an activity may spawn multiple activities in the same place or in other places. Further it may wait until all activities spawned during the course of execution of a statement have (recursively) terminated, using the `finish S` construct. Activities may use *conditional atomic blocks* for synchronization (`when (c) S`). Such a block permits an activity to wait until a specified condition `c` is true, and then in one atomic step execute the statements in `S` (in the same state in which `c` is true). Conditional atomic blocks are a very simple and powerful synchronization construct, and can be used to elegantly implement all the usual concurrency co-ordination idioms (producer/consumer synchronization, mutual exclusion, barrier synchronization etc). Of particular interest in X10 are *barriers*

Atomic blocks are supported, with implicit locking.

A.2 Messaging Passing Interface (MPI)

MPI is a popular two-sided messaging passing communication support. MPI-2 adds one-sided extensions, albeit with complex semantics. MPI is still primarily used as a two-sided library. It is available on almost all high performance computing platforms. Inter-operability with MPI can ease user acceptance of X10lib. In addition, X10lib uses the process management framework from MPI, i.e., an X10lib program is started as an MPI program.

A.3 LAPI

LAPI (Low-level API) [1] provides efficient remote memory access mechanisms (one-sided communication). The LAPI library provides basic operations to "put" data to and "get" data from one or more virtual addresses of a remote task. LAPI also provides an active message infrastructure. With active messaging, programmers can install a set of handlers that are called and run in the address space of a target task on behalf of the task originating the active message. These handlers can be used to dynamically determine the target address (or addresses) where data from the originating task must be stored.

LAPI may operate either in *polling* or *interrupt* mode. In the former, a user pthread invoking a LAPI function may be temporarily borrowed to send or receive messages. In the latter a dedicated single pthread (created at initialization time) is responsible for handling message send/receive traffic.

A.4 ARMCI

ARMCI is a portable communication library that provides one-sided communication facility. It provides the functions for memory to memory transfer operations, accumulation, read-modify-write operations, memory allocation etc. The data transfer operations are available in the form of two noncontiguous operations : ARMCI_PutV and ARMCI_PutS. ARMCI_PutV uses general I/O vectors to describe the source and destination memory locations. This is useful for transferring any kind of data, with even non-constant stride (e.g. triangular section of an array). ARMCI_PutS is useful for transferring strides region with constant strides. For more information on these function refer to [4].

ARMCI also provides two atomic operations : *accumulate* and *read-modify-write*. Accumulate operation combines the local and remote data atomically : $x = x + a \times y$. Read-modify-write updates a remote `integral` variable according to a specified operation and returns the old value.

ARMCI also provides a simple progress and ordering rules. The progress rule is that all the ARMCI one-sided operations complete regardless of the actions taken by the receiver. That is, there is no need for the remote process to make occasional communication calls or poll in order to assure that communication calls issues by other processes to this process can complete.

The ARMCI operations issued to the same destination process complete in order. Operations issued to different processors can complete in an arbitrary order. Additionally, when a put or accumulate operation completes, the data has been copied out of calling process memory but has not necessarily arrived at its destination. This is a local completion. A global completion can be achieved by called ARMCI_Fence operation.

A.5 GASNet

GASNet (Global-Address Space Networking) is a network-independent and language-independent high-performance communication interface for use in implementing the runtime system for global address space languages. It provides two layers of interface - core and extended. The core API is a narrow interface based on the Active Messages paradigm. The extended API is an expressive and flexible interface that provides medium and high-level operations on remote memory and collective operations.

The core API provides the active messages functionality. Active message communication is formulated as logically matching request and reply operations. Upon receipt of a request message, a request handler is invoked; likewise, when a reply message is received, the reply handler is invoked. Request handlers can reply at most once to the requesting node and only to the requesting node. The active message routines are divided in to three kinds based on the size of the message transfer : gasnet_AMRequestLongM(), gasnet_AMRequestMedimM() and gasnet_AMRequestShortM(). Corresponding reply routines are also provided. Additionally, an asynchronous version of long request is provided in the form of gasnet_AMRequestLongAsyncM().

A key feature in the active message interface of GASnet is the ability for programmer to supply several parameters to AM handlers, as a part of the AM message routines. The library internally communicates those parameters to the receiver. This functionality is useful for implementing remote *async* activities. A remote *async* activity is just a function call that has to executed on a place P. An *async* can also access certain local variables (e.g. final variables) from the parent's local stack. These variables can be passed as arguments to the function call.

The extended API provides memory to memory transfer operations in the form of put and get. Both blocking and non-blocking versions are provided. The non-blocking versions do not complete the operation, but return a handle. A synchronization operation on the handle should be performed to complete the operation. Additionally, register-to-memory transfer operations are also provided.

B Notes for X10 compiler writers

B.1 Garbage collection

B.2 Asyncs

B.3 Atomics

B.4 Arrays

References

[1] Rsct for aix 5l: Lapi programming guide. Technical Report SA22-7936-05, IBM, october 2006.

[2] Dan Bonachea. Gasnet specification, v1.1. Technical Report UCB/CSD-02-1207, CS Division, EECS Department, University of California, Berkeley, october 2002.

[3] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[4] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The armci approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, 2006.

[5] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.