The X10lib API

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 14, 2007

Abstract

This document presents a snapshot of the API for X10lib.

1 Introduction

2 API

3 X10 RTS API Definition

3.1 Scenario

Each X10 place is executed in exactly one X10 process. An X10 place can have multiple OS threads for activity computation. The X10 runtime is invoked using an `initialize` routine and terminated using `finalize` routine. The X10 runtime starts the execution of the application program in a boot activity which acts as the global termination point for subsequent parallel computations in the application program.

Each X10 place apart from having a bunch of OS threads, maintain a Doubly-Ended Queue (Deque) of activities. Activities can be added on either-end but can only be removed from the bottom. For the rest of our discussion, we will assume the Deque to be unbounded.

An activity can be "blocking" or "non-blocking" based on the kind of operation it performs in its body. Note that `atomic` constructs in our implementation are non-blocking; `finish`,`force()` and `clock.next()` are blocking operations. During compiler analysis phase if we can find an activity to be non-blocking, we will try and inline many of such non-blocking activities as if they were sub-routine calls from their repective parent activities. If inlining is not possible, we will annotate the activity creation with non-blocking flags which can then be used by the runtime for additional inlining option. *There may be some exception to the extent of inlining – for example* `foreach` *construct iterating over a huge region – we may not want to get rid of intra-place parallelism completely if the body of* `foreach` *is non-blocking.*

We define the term *finish-proxy* for an activity as follows: the finish proxy for an activity is the activity containing the innermost finish scope. All the application level activities which are not created within any finish scope will have the boot activity as their finish-proxy.

The boot activity is resident at 0th Place and is immediately picked by OS threads for computation. The boot activity can:

- Create a local child activity (`async`): Using the compiler annotations, if we can determine the child activity to be "non-blocking", we can execute the body of the activity as if it were a sub-routine call. If the local child activity is blocking, the child activity is added to the bottom of the deque at the local place. The parent activity continues its execution.

- Create a remote child activity (`async (P)`): The activity is added to the top of the deque at the remote place. The parent activity continues execution.
- Executes `finish` block: The parent activity marks itself as the finish-proxy for any subsequent child activity created in the `finish` scope. The body of the `finish` block is implemented using *protothreads* and *co-routines* – certain constraints exist (Look at Sriram's design x10lib/docs/x10lib-design.pdf – page 4). Any child remote activity created within the body of the `finish` block will have to report its termination to its finish-proxy. Instead of sending individual finish-notifications, we can club multiple termination reports for the same place (optimizations possible).
- Executes `future-force` block: If the future block is targetted to the local place and is non-blocking, we will inline the body of the future block. Otherwise, the future block is implemented using *protothreads* and *co-routine* as like `finish`
- Executes `atomic` block: Use specialized non-blocking instructions on SMP to implement `atomic`.
- Perform clock non-blocking operations: standard interfaces needed for `register`, `drop`, `resume`, `isRegistered`.
- Perform clock blocking operation (`next`): Use similar technique of *protothreads* and *co-routine* to perform suspend and enablement.

3.2 Declaration and Initialization

The X10 runtime header for the translation:

- `<x10\_rts.h>`

  This contains the description of all data structures for activities, places, clocks, states, stack, activity deque, etc. and the `extern` interfaces mentioned below. Many of the interfaces have `int` as return type for returning *error_type_t*.

3.2.1 Activity Spawn

Assumption: There exists an activity object with pointers to its place of execution.
X10 *syntax:*

```
async (place) S;
```

C syntax:

```
#include ‘‘x10_rts.h’’

int activity_dispatch (place_t place, activity_t child, activity_t parent, bool_t may_block, int (*acti
```

Description:
The command is invoked from the parent activity. It decides either to inline the child activity or not based on the flag `may_block`. If it decides not to inline, it adds the body of the activity to the destination places deque (local place – bottom of deque; remote place – top of deque).

3.2.2 Activity Block

X10 *syntax:*

- `force`
- `next`

- `finish`

C syntax:

```
#include ‘‘x10_rts.h’’

int activity_block_start (place_t place, activity_t a, data_t block_data_structure);
int activity_block_stop (place_t place, activity_t a, data_t block_data_structure);
```

Description: Use the techniques of *protothreads* and *co-routines* to implement blocking semantics in X10.

3.2.3 Activity Completes

C syntax:

```
#include ‘‘x10_rts.h’’

int activity_finish (place_t place, activity_t a, activity_t innermost_finish_scope_activity};
```

Description: Send a finish notification to the finish-proxy. Propagate any exception.

3.2.4 Clock Registration

X10 *syntax:*

- `new clock()`
- `async (place) clocked c1,c2 S;`

C syntax:

```
#include ‘‘x10_rts.h’’

int register_clock (activity_t a, clock_t *c);
bool is_registered (activity_t, clock_t *c;
```

Description: Register the activity with the set of clocks if not already done. Note that the clock objects are already created.

3.2.5 Clock Deregistration

X10 *syntax:*

- `c1.drop()`

C syntax:

```
#include ‘‘x10_rts.h’’

int deregister_clock (activity_t a, clock_t *c);
```

Description: Deregister the activity with the set of clocks explicitly.

3.2.6 Clock Resume

X10 *syntax:*

- `c1.resume()`

C syntax:

```
#include ‘‘x10_rts.h’’

int resume_clock (activity_t a, clock_t *c);
```

Description: Resumption on a set of clocks for an activity.

3.2.7 Clock quiescence

X10 *syntax:*

- `next()`

C syntax:

```
#include ‘‘x10_rts.h’’

int next_phase_clock (activity_t a, clock_t *c);
```

Description: Perform clock quiescence using the blocking technique described above.

3.3 API for PGAS and Active Messages

The methods in this API implement the partitioned global address space (PGAS) and provide remote memory access either using put and get operations or active messages. The methods are inspired from GASnet [1] and ARMCI [2].

3.3.1 GAS Initialization and Management

We provide the following methods related to GAS:

```
int gas_init (...);

int gas_finish (...);

int gas_alloc (int nbytes);

int gas_free (gas_addr addr);
```

3.3.2 Active Messages

The active message API is inspired from GASnet. A key feature in the active message interface of GASnet is the ability for programmer 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. Another feature of GASnet is that it provides three different types of AM targeting short, medium and long messages. This functionality will be also provided in the X10lib; X10compiler can recognize the size of the remote messages (e.g. a[i] = x vs a[1:1000] = x[1:1000]) and use the appropriate method to handle communication.

There are three kinds of methods in this category. All the methods are appended with a number M. M should be replaced by a number between $0 to NUMARGS - 1$ in the actual call; $NUMARGS$ is a pre-defined constant that specifies the maximum number of arguments allowed for a handler. All the calls are non-blocking. That is the sender does not wait for the completion of the receipt at the receiver end. However, the source buffer may be safely over-written. That is, on the sender side the message is guaranteed to be sent (or buffered). As a future extension, we can also consider *asynchronous* versions of the active message send operations, which do not wait for the message to be sent (or buffered).

- i) :

```
int am_send_shortM (int dst, handler_t handler, exec_type_t e,
                       arg_type_t arg0, arg_type_t arg1, ... arg_type_t argM-1)
```

- ii) :

```
int am_send_mediumM (int dst, handler_t handler, exec_type_t e, void* source_addr, int nbytes,
                         arg_type_t arg0, arg_type_t arg1, ... arg_type_t argM-1)
```

- iii) :

```
int am_send_longM (int dst, handler_t handler, exec_type_t e, void* source_addr, void* dst_addr, in
          arg_type_t arg0, arg_type_t arg1, ... arg_type_t argM-1)
```

- iv) :

```
int am_wait (handler_t handler);
```

3.3.3 Memory transfer operations

Though the active messages are general enough to transfer values from a local memory to remote memory it is desirable to have a specialized memory transfer operations for various reasons. Chief reason is active messages are not sufficient to transfer non-contiguous data efficiently. Additionally, functionalities like aggregate remote copy can be provided in this API.

All the methods in the category take the following three parameters: pointer to destination memory, pointer to source memory and number of bytes to be transferred. Optionally, pointer to a `stride_desc_t` can be passed as the fourth argument. `stride_desc_t` is a structure with `dst_stride_arr`, `src_stride_arr`, dim and count as its primary member. `dst_stride_arr` and `src_stride_arr` describe the stride of the data in destination and source memory for each dimension of the array. dim specifies the number of dimensions of the array and count specifies the size along each dimension. If `stride_desc` is null, the data is assumed to be contiguous.

Another optional argument is the destination node that can be passed as the last argument. This is not required for 64-bit architectures, but only for 32-bit architectures.

We provide three kinds of put and get operations : blocking, non-blocking and aggregate. Calls to blocking put and get operations block until the transfer is complete. Non-blocking put and get methods do not block until the transfer is complete. Instead, they return immediately with a handle. The transfer is completed only on a `sync` operation on the handle. However, the source buffer is safe to be re-used after the put and get methods return. The contents of the destination memory is un-defined until the synchronization completes successfully. Aggregate puts and gets queue all the non-blocking puts and gets on a given handle and finish them at once during a synchronization operation on that handle.

- i) :

```
int put (void* dst, void* src, int nbytes,
 stride_desc_t *stride_desc = NULL,
         int dest = -1)

handle_t put_nb (void* dst, void* src, int nbytes,
    stride_desc_t *stride_desc = NULL,
            int dest = -1)

handle_t put_nb_agg (void* dst, void* src, int nbytes,
    stride_desc_t *stride_desc = NULL,
            int dest = -1)
```

- ii) :

```
int get (void* dst, void* src, int nbytes,
 stride_desc_t *stride_desc = NULL,
      int dest = -1)

handle_t get_nb (void* dst, void* src, int nbytes,
         stride_desc_t *stride_desc = NULL,
       int dest = -1)

handle_t get_nb_agg (void* dst, void* src, int nbytes,
stride_desc_t *stride_desc = NULL,
             int dest = -1)
```

- iii) :

```
int sync (handler_t handle)

int sync_all()
```

3.3.4 Read-Modify-Write Operatons

X10lib also provides atomic read-modify-write operations, that atomically updates a remote location with a given value and operation.

```
int rmw (void *dst, long value, int op, int dest = -1)

int rmw_nb (void *dst, long value, int op, handler_t handle, int dest = -1)
```

3.3.5 Collective Operations

The following collective Operatons are provided by X10lib : `broadcast`, `reduce` and `allreduce`.

- i):

```
void broadcast (void* data, int nbytes, int root=-1)
```

- ii):

```
void reduce (void* data, int nbytes, int type, op_t op, int root)
void allreduce (void* data, int nbytes, int type, op_t op)
```

- iii):

```
void barrier(void)
```

3.4 Array

```
template<int N>
class Point
{
 private:
 const int values_[N];
};

class point<1>
{
   const int i_;
};

class point<2>
{
   const int i_;
   const int j_;
};

class point<3>
{
   const int i_;
   const int j_;
   const int k_;
};


template<int RANK>
class Region
{
  public:

  int linearIndex (const Point<RANK>& x) const;
  bool isEqual (const Region<RANK>& r) const;

};


template<int RANK>
class ConvexRegion : private Region<RANK>
{
  public:

  Region (int size[RANK], int stride[RANK], const Point<RANK>& origin);

  int linearIndex (const Point<RANK>& x) const;

  bool isEqual (const Region<RANK>& r) const;

  private:

  int size_[RANK];

  Point<RANK> origin;

  int stride_ [RANK];

  int linearStep_ [RANK];


  //for regions that are formed by restriction operation
  int globalLinearStep_ [RANK];

};

template<int RANK>
class ArbitRegion : private Region<RANK>
{
  public:

  int linearIndex (const Point<RANK>& x) const;

  bool isEqual (const Region<RANK>& r) const;

  private:

  Region<Rank>* regions_;
  int numRegions_;

};


template<int RANK>
```

```
Region<RANK> intersect (const Region<RANK>& r1, const Region<RANK>& r2);

template<int RANK>
Region<RANK> union (const Region<RANK>& r1, const Region<RANK>& r2);

template<int RANK>
Region<RANK> difference (const Region<RANK>& r1, const Region<RANK>& r2);


template <int RANK>
class Distribution
{
public:

protected

   Region<RANK> region;

};

template <int RANK>
class ConstDistribution : private Distribution<RANK>
{

public:


};

template <int RANK>
class BlockCyclicDistribution : private Distribution<RANK>
{
public:

};

template <int RANK>
class UniqueDistribution : private Distribution<RANK>
{
public:

};


template <typename T, int RANK>
class Array
{
Array (Distribution<RANK> dist);

Array (Array<T, RANK>& A, Region<Rank>& R);
Array<T, RANK>* clone();

Distribution<RANK> dist() const;

        T getScalarAt (const Point<RANK>& P) const;

T getScalarAt (int n) const;

        void putScalarAt (const Point<RANK>& P, T val);

void putScalarAt (int n, T val);

~Array();

private:

T* data_;

protected:

        Distribution <RANK> dist_;
};

// Useful for casting a scalar to array

template <typename T, int RANK>
class UnitArray : public Array <T, RANK>
{
   UnitArray (int value) : Array();

   T& operator[] (const Point<RANK>& P);

   Array<T, RANK>& operator[] (const Region<RANK>& R);

   private:

   T value_; //the same value is replicated in the array
     //upon a write, create a new data_ for this array
};

//initialization routines
template <typename T, int RANK, typename CONST_INIT>
void initialize (Array<T, RANK>& arr, CONST_INIT op);

template <typename T, int RANK, typename POINT_INIT> (check if this is valid)
void initialize (Array<T, RANK>& arr, POINT_INIT<RANK> op);

//pointwise routines for standard operators
template <typename T, int RANK>
void iterate (Array<T, RANK>& arr, order_t order, op_t op);
```

```
template <typename T, int RANK, int N>
void iterate (Array<T, RANK> (&args) [N], order_t order, op_t (&op)[N]);

//pointwise routines for "lift"ed operators
template <typename T, int RANK, typename SCALAR_OP>
void iterate (Array<T, RANK>& arr, order_t order, SCALAR_OP op);

template <typename T, int RANK, int N, typename SCALAR_OP>
void iterate (Array<T, RANK> (&args) [N], order_t order,  SCALAR_OP op);

//reduce
template <typename T, int RANK-1>
void reduce (Array<T, RANK> &arg, int dim, op_t op);

//scan
template <typename T, int RANK-1>
void scan (Array<T, RANK> &arg, int dim, op_t op);


//restriction
Array<T, RANK>& restriction (const Distribution<RANK>& R);


//assembling

Array<T, RANK>& assemble (const Array<T, RANK>& a1, const Array<T, RANK>& a2);

Array<T, RANK>& overlay (const Array<T, RANK>& a1, const Array<T, RANK>& a2);

Array<T, RANK>& update (const Array<T, RANK>& a1, const Array<T, RANK>& a2);

// How are value arrays and reference arrays reflected in the design?
//     o In the library it is always reference; the compiler should create copies for value arrays.
```

References

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

[2] 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.