Multi-place X10 programs (v 0.1)
Vijay Saraswat

1 Introduction

Quick references.

- Read Chapter 13 (Places) in [SBP+].
- Familiarize yourself with the APIs in the following standard libraries. In `x10.lang`, look at `Place`, `PlaceGroup`. In `x10.util` look at `Team`. In `x10.array`, look at `Dist*` classes. Look at the implementation of the `x10.array` classes. It is not difficult to design your own distributed array data-structures, with the appropriate distributions.
- In the X10 SVN (on sourceforge) look in the `courses / pppp2013` repository. `AllReduceTeams.x10` shows how to use `DistArray_Unique`. `NQueensDist` shows the collecting `finish` idiom, working across multiple places.
- Example multi-place programs that run at scale: In the benchmarks / PERCS directory:

  STREAM / Stream.x10: a simple program measuring bandwidth from the memory system.

  KMEANS/KMeans.x10: implements an interative multi-place algorithm for computing K clusters for a given set of data-points.

  FT/FTNew.x10: implements a multi-place FFT algorithm, using three AllToAll transpositions.

  LU/LUNB.x10: Implements LU decomposition of a matrix.

  GLB-UTS-SSCA2: This implements the "global load balancing" framework. This is a library that lets you specify tree-structured tasks are look at glb/GlobalLoadBalancer.x10, bcg/BCG.x10 and uts/UTS.x10.
- Many sections in "A Brief introduction to X10" (see course website) are relevant. Specificaly, Section 4 (the X10 Performance Model) and Part II (Programming Examples). In Part II, KMeans is covered in Section 6.3, FU in 6.4 and LU in 6.5.

There are two fundemantal reasons to go to multiple places. The data may be much larger than can fit into one place. And/or the amount of computation to be performed may be significantly higher, hence more cores (hundreds, thousdans, ...) will help.

The fundamental problem in going to multiple places is that the programmer has to be aware of *locality*. The *latency* and *bandwidth* of data access changes dramatically when one goes off-place. Latency refers to the delay between issuing a memory access and obtaining a value. Bandwidth refers to the total amount of data that can be transferred in unit time from the destination to the processing unit. X10 does not attempt to deal "silently" with these latency and bandwidth disparities – instead it surfaces these issues into the programming model by introducing a notion of *places*. A place has data (a heap) and multiple activitiies that may operate on that data. A single computation may run over multiple (hundreds, thousands) of places. It is the programmer's responsibility to divide up the data among the given places and take into account latency and bandwidth issues when writing code to operate on this data.

2 At statements and expressions

The fundamental control construct available in X10 for multi-place computation is the `at` construct.

An `at` statement takes the form `at (p) S`, where `p` is an expression that evaluates to a place, and `S` is a statement. Such code may be executed at any place `q` (`q` may be the same as `p`).

`S` is permitted to contain occurences of variables defined outside the scope of `S`. At runtime, the entire graph G of objects accessible from the free variables of `S` is serialized into a buffer, transported to `p`, deserialized there to reconstruct an object graph G' isomorphic to G, the free variables of `S` are bound to the corresponding objects in G', and then `S` is executed in a separate activity. Once `S` terminates locally, control returns to `q` and continues with the next statement after the `at(p) S`.

Note that at `p`, `S` may spawn multiple activities. These will be controlled by the `finish` controlling the `at(p) S`, as is usual for local termination. Note further that `S` may throw an exception. In this case `at(p)S` will throw the same exception. Thus exception throwing can cross boundaries.

An `at` expression takes the form `at (p) {e}`: it goes to `p`, evaluates `e`, and returns the value.

Note: A primary mistake programmers make is to refer to too much state

in `S`. *All* state referenced in `S` will be copied over. You can set run-time flags to tell you how many bytes are shipped over on an `at`. This can help you catch the problem. Also in X10DT if you move the cursor over an `at` statement, you will be shown the variable which are captured within the `at`. This can be very valuable in performance debugging.

2.1 Place failure and Resilient X10

Long running jobs running in a commodity cluster may fail because a place fails – either a disk fails, or there is a network outage, or, sometimes, there is planned or unplanned maintenace of the cluster.

An X10 computation will fail if any place fails. The X10 launcher monitors each process (place) that is launched in the computation. If it is unable to determine that a place is up, it will send signals to all remaining places to tear them down, thus aborting the computation.

In recent work, the X10 team has developed a notion of *Resilient X10* [**?**]. A place is permitted to fail without bringing down the entire computation. Instead, the data at the failed place `q` and the activities operating on it, are lost. Any activity `a` launched by code at `q` at other places `r` continue to run, and are now controlled by the "closest" (in dynamic scope) synchronous statement (e.g. `finish`) not running at `q` that contains `a`. X10 provides a "Happens Before Invariance" guarantee: the happens before relation between steps that remain after a place failure is the same as the relation for the original program. This considerably simplifies programming resilient computations in X10. Any attempt made by a place `p` to execute an `at` results in a `DeadPlaceException` that may be caught by user code.

Resilient X10 has not yet been released, the implementation is under development at `x10.sf.net` and may be downloaded from SVN.

3 Global shared state

3.1 x10.lang.GlobalRef

3.2 x10.lang.PlaceLocalHandle

4 x10.lang.PlaceGroup

4.1 SPMD computations

Many programs can be parallelized in a *data parallel* fashion.

5 Data distribution

Need to understand patterns of communication and computation to determine how best to partition data.

Commonly, arrays. Multi-dimensional arrays.

5.1 Row block

5.2 Block cyclic distribution

6 FT

7 LU

8 The GLB library

The GLB (Global Load-Balancing) framework is applicable for *tree-structured problems*.

Such problems can be characterized thus. There is an initial bag (multiset) of *tasks* (may be of size one). A task usually has a small amount of associated state, but is permitted to access (immutable) "reference state" that can be replicated across all places. Consequently, tasks are *relocatable*: they can be executed at any place.

Tasks can be executed. Executing a task may only involve local, self-contained work (no communication). During execution, a task is permitted to create zero or more tasks, and produce a result of a given (pre-specified) type `Z`. The user is required to specify a reduction operator of type `Z`.

The GLB framework is responsible for distributing the generated collection of tasks across all nodes. Once no more tasks are left to be executed, the single value of type `Z` (obtained by reducing the results produced by all tasks) is returned.

Since the execution of each task depends only on the state of the task (and on immutable external state), the set of results produced on execution will be the same from one run to another, regardless of how the tasks are distributed across places. Since the user supplied reduction operator is assumed to be associative and commutative, the result of execution of the problem is determinate. Thus GLB is a determinate application library that requires the user to provide a few sequential pieces of code and handles concurrency, distribution and communication automatically.

The GLB framework is applicable to a wide variety of tasks. For a simple example, consider the problem of computing the n'th Fibonacci number in

the naive recursive way. Here the state of the task can be summarized by just one number (long), i.e. n. Execution of the task yields either a result (for $n < 2$), or two tasks (for $n - 1$ and $n - 2$). The results from all tasks need to be reduced through arithmetic addition.

All *state space search algorithms* from AI fall in this category. Such algorithms are characterized by a space of states, an initial state, and a generator function which given a state generates zero or more states. The task is to enumerate all states (perhaps with a cutoff), apply some function, and some up the results. An example of such an application is the famous N-Queens problem.

Another example of this problem is the Unbalanced Tree Search (UTS) benchmark. Indeed, it was our attempt to solve the UTS problem that led to the creation of the GLB framework.

8.1 The GLB API

The GLB framework works by distributing bags of tasks from busy places to idle places automatically. In special cases, the application programmer may improve overall performance by specifying a custom implementation of `TaskBag`. The implementation must support a `split` operation which permits the current task bag to be split into two (so the other piece can be distributed to another place) and a `merge` operation which permits incoming work to be included in the task bag at the current place.

Thus in summary, the application programmer provides the following pieces of code:

1. A `TaskFrame[Z]` class. This class is expected to maintain a reference to a `TaskBag` (the *local task bag*). The user must supply implementations for the following methods:
   (a) `getTaskBag():TaskBag`: should return the local task bag.
   (b) `getReducer():Reducible[Z]`: should return the reducer associated with this job.
   (c) `getResult():Z`: This returns the accumulated result for all tasks that were completed at this place.
   (d) `initTask():void`: This is called once, when this task frame is set up at the current place. It should initialize the local task bag appropriately.

If the problem has a single root task, then the user should choose one place (e.g. place zero) as the place at which to add this task. If the problem has many initial tasks then more than one place may start out with non-empty task bags.

(e) `runAtMostNTasks(n:Long):Boolean`: This method must run at most `n` tasks from the task bag at the current place, and return `true` if there is still work left to do in the local task bag. During the execution of this method, user code may manipulate the local task bag freely (removing and adding tasks to it). The framework guarantees that no other worker will concurrently access the task bag. For tasks which have completed and produced a result of type `Z`, the method must accumulate the result in local state, using the supplied reducer.

2. Optionally, a custom class extending `TaskBag` and appropriately implementing `split` and `merge` methods.

The framework is invoked in a fairly simple way:

```
val r = new GlobalLoadBalancer[T](parameters);
val result = g.run(()=> new TaskFrameImp[T]());
```

First an instance of `GlobalLoadBalancer` is created with a specification of parameters for controlling various aspects of global load-balancing. Next the framework is invoked, supplying a function that generates an instance of the user-specified `TaskFrame` class, with the appropriate result type. This method returns the result of running the user's problem, balanced across all places.

The code in Figure 1 illustrates the use of this framework with a simple Fibonacci computation.

9 UTS

9.1 UTS description

The Unbalanced Tree Search benchmark measures the rate of traversal of a tree generated on the fly using a splittable random number generator. Below, to fix discussion, assume we are working with a *fixed geometric law* for the random number generator with branching factor $b_0 = 4$, seed $r = 19$, and tree depth d varying from 13 to 20 depending on core counts and architecture.

```
package fib;

import glb.TaskBag;
import glb.TaskFrame;
import glb.GlobalLoadBalancer;
import glb.ArrayListTaskBag;
import glb.GLBParameters;

public class Fib(n:Long) {
    class FibFrame extends TaskFrame[Long] {
        val bag = new ArrayListTaskBag[Long]();
        val reducer = Reducible.SumReducer[Long]();
        var result:Long=0;
        public def getTaskBag()=bag;
        public def getReducer()=reducer;
        public def getResult()= result;
        public def initTask() {
            if (here.id==0) bag.bag().add(n);
        }
        public def runAtMostNTasks(n:Long) {
            val b = bag.bag();
            for (1..n) {
                if (b.size()<=0) break;
                val x = b.removeLast(); // constant time
                if (x < 2) result += x;
                else {
                    b.add(x-1); // constant time
                    b.add(x-2);
                }
            }
            return b.size()>0;
        }
    }
    public def run():Long {
        val g = new GlobalLoadBalancer[Long](GLBParameters.Default);
        return g.run(()=>new FibFrame());
    }
    static def fib(n:Long):Long=n<2? n: fib(n-1)+fib(n-2);
    public static def main(args:Rail[String]) {
        val N = args.size < 1 ? 10 : Long.parseLong(args(0));
        val result = new Fib(N).run();

        Console.OUT.println("fib(" + N + ")="
           + result + " (actual=" + fib(N)+")");
    }
}
```

Table 1: Fibonacci, with GLB

> The nodes in a geometric tree have a branching factor that follows a geometric distribution with an expected value that is specified by the parameter $b_0 > 1$. The parameter d specifies its maximum depth cut-off, beyond which the tree is not allowed to grow ... The expected size of these trees is $(b_0)^d$, but since the geometric distribution has a long tail, some nodes will have significantly more than b_0 children, yielding unbalanced trees.

The depth cut-off makes it possible to estimate the size of the trees and shoot for a target execution time. To be fair, the distance from the root to a particular node is never used in our benchmark implementation to predict the size of a subtree. In other words, all nodes are treated equally, irrespective of the current depth.

9.2 Design

The material in this section is excerpted from [**?**].

One common way to load balance is to use *work-stealing*. For shared memory system this technique has been pioneered in the Cilk system. Each worker maintains a double ended queue (deque) of tasks. When a worker encounters a new task, it pushes its continuation onto the bottom of the deque, and descends into the task. On completion of the task, it checks to see if its deque has any work, if so it pops a task (from the bottom) and continues with the work. Else (its deque is empty) it looks for work on other workers' deques. It randomly determines a *victim* worker, checks if its queue is non-empty, and if so, pops a task from the top of the deque (the end other than the one being operated on by the owner of the deque). If the queue is empty, it guesses again and continues until it finds work. Work-stealing systems are optimized for the case in which the number of steals is much smaller than the total amount of work. This is called the "work first" principle – the work of performing a steal is incurred by the thief (which has no work), and not the victim (which is busy performing its work).

Distributed work-stealing must deal with some additional complications. First, the cost of stealing is usually significantly higher than in the shared memory case. For many systems, the target CPU must be involved in processing a steal attempt. Additionally, one must solve the distributed termination detection problem. The system must detect when all workers have finished their work, and there is no more work. At this point all workers should be shut down, and the computation should terminate.

X10 already offers a mechanism for distributed termination detection – `finish`. Thus, in principle it should be possible to spawn an activity at each place, and let each look for work. To trigger `finish` based termination detection however, these workers must eventually terminate. But when? One simple idea is that a worker should terminate after k steal attempts have failed. However this leaves open the possibility that a worker may terminate too early – just because it happened to be unlucky in its first k guesses. If there is work somewhere else in the network then this work can no longer be shared with these terminated workers, thereby affecting parallel efficiency. (In an extreme case this could lead to sequentialize significant portion of the work.)

Therefore there must be a way by which a new activity can be launched at a place whose worker has already terminated. This leads to the idea of a *lifeline graph*. For each place p we pre-determine a set of other places, called *buddies*. Once the worker at p has performed k successive (unsuccessful) steals, it examines its buddies in turn. At each buddy it determines whether there is some work, and if so, steals a portion. However, if there is no work, it records at the buddy that p is waiting for work. If p cannot find any work after it has examined all its buddies, it dies – the place p now becomes quiescent.

But if P went to a buddy B, and B did not have any work, then it must itself be out looking for work – hence it is possible that it will soon acquire work. In this case we require that B *distribute* a portion of the work to those workers that attempted to buddy steal from it but failed. Work must be spawned on its own `async` – using the `at (p) async S` idiom. If P had no activity left, it now will have a fresh activity. Thus, unlike pure work-stealing based systems, a system with lifeline graphs will see its nodes moving from a state of processing work (the active state), to a state in which they are stealing to a state in which they are dead, to a state (optionally) in which they are woken up again with more work (and are hence back in the active state).

Note that when there is no work in the system, all places will be dead, there will be no active `async` in the system and hence the top-level `finish` can terminate.

The only question left is to determine the assignment of buddies to a place. We are looking for a directed graph that is fully connected (so work can flow from any vertex to any other vertex) and that has a low diameter (so latency for work distribution is low) and has a low degree (so the number

of buddies potentially sending work to a dead vertex is low). z-dimensional hyper-cubes satisfy these properties and have been implemented.

9.3 UTS-G

UTS-G is a variant of UTS written using the GLB library. The code for `GlobalLoadBalancer` and `LocalJobRunner` is no longer needed, since they are part of the library. The code for managing bags of tasks is refactored into a `UTSTaskBag` class. Similarly the code for executing a UTS task `runATMostNTasks` in `UTSTaskFrame` is similarly very straightforward. The native code (for SHA1 hashes) remains unchanged – it belongs in the application and not the library.

Thanks to the GLB library the number of X10 LOCs drops from 493 for UTS to 225 for UTS-G.

10 Betweenness Centrality

10.1 Problem description

The Betweenness Centrality benchmark is taken from the SSCA2 (Scalable Synthetic Compact Application 2) v2.2 benchmark [**?**]; specifically, we implement the "fourth" kernel in this benchmark, the computation of *betweenness centrality*:

> The intent of this kernel is to identify the set of vertices in the graph with the highest betweenness centrality score. Betweenness Centrality is a shortest paths enumeration-based centrality metric, introduced by Freeman (1977). This is done using a betweenness centrality algorithm that computes this metric for every vertex in the graph. Let σ_{st} denote the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ the number of those paths passing through v. Betweenness Centrality of a vertex v is defined as $BC(v) = \Sigma_{s \neq v \neq t \in V} \sigma_{st}(v)/\sigma_{st}$.
>
> The output of this kernel is a betweenness centrality score for each vertex in the graph and the set of vertices the highest betweenness centrality score.

We implement the Brandes' algorithm described in the benchmark, augmenting Dijkstra's single-source shortest paths (SSSP) algorithm, for unweighted graphs. We have implemented the exact variant of the benchmark (`K4approx=SCALE`).

The solution we implement makes one very strong assumption: that the graph is "small" enough to fit in the memory of a single place. Since the graph itself is not modified during the execution of this benchmark, it thus becomes possible to implement this benchmark by replicating the graph across all places. Now effectively the computation can be performed in an embarrassingly parallel fashion. The set of N vertices is statically partitioned among P places; each place is responsible for performing the computation for the *source* vertices assigned to it (for all N target vertices) and computes its local `betweennessMap`. After this is done an `allReduce` collective performs an `AND` summation across all local `betweennessMaps`.

Clearly, statically partitioning the work amongst all places in this way is not ideal. The amount of work associated with one source node v vs another w could be dramatically different. Consider for instance a degenerate case: a graph of N vertices, labeled $1 \ldots N$, with an edge (i, j) if $i < j$. Clearly the work associated with vertex 1 is much more than the work associated with vertex N.

Section **??** discusses how to dynamically load-balance these tasks across all places using the GLB framework (Section **??**).

10.2 Structure of the code

The multi-place structure of this code is quite straightforward. We use the standard at-scale X10 idiom for SPMD code:

1. Create a distributed data structure, accessed through a `PlaceLocalHandle` which points to an instance of a "host" object (an instance of `SSCA2`) at each node. (Invocation of `makeFlat`.)

2. Run the actual computation (invocation of `broadcastFlat` with the closure specifying the work to be done at each place.

   The work done in each place is the computation of a `bfsShortestPaths` for a slice of vertices from `startVertex` to `endVertex`. This loops through the vertices in the given range, computing the betweennessMap locally.

   Note this computation is single threaded. It is not difficult to make this multi-threaded.

3. Once the local computation has been done, and the local contribution to the betweennessMap computed, a `Team.WORLD.allreduce` ensures that the results are reduced and made available at place 0.

10.3 BC-G

BC-G implements dynamic load-balancing by using the GLB library. Note that this is a new capability (over what is supported by BC) – therefore it is not surprising that new code needs to be written and the line count is actually higher than for BC. Specifically, `BCTaskFrame` and `BCTaskBag` needs to be implemented, together with the invocation of the `GLB` framework. A task captures a (start, end) range of vertices. The code for `BCTaskFrame` is straightforward – it pops tasks from the `BCTaskBag` and for each task executes it by calling `bfsShortestPaths` already implemented in BC. The reducer takes two betweenness maps (vectors) and sums them up pointwise.

We now consider the implementation of `BCTaskBag`. Our current implementation (reported in this submission) is preliminary and in need of further work. It is designed to satisfy the following properties. It should be possible to find the "mid point" fast so that the bag can be split in two quickly. The cost of a merge should be low.

The current implementation uses an array, a head pointer and a tail pointer. The head pointer is moved forward whenever the task frame retrieves an item and works on it. The tail pointer is used during splitting, so it can move to the split point, copy the latter half of the data and send it out.

When merging, we create a new array and copy the local taskbag and the incoming taskbag to the new array and set up the head and tail pointer properly. We believe this copy is expensive and should be eliminated.

We believe a more careful design for this data-structure will improve performance. Basically we need to design a data-structure that represents a set of intervals such that two such sets can be combined quickly and split quickly, where the split ensures that the sum of the sizes of intervals in each half are roughly equal. We believe that improving the design of this data-structure will significantly improve performance. *We expect to work on this in the next few weeks, before SuperComputing.*

References

[SBP+] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.