Solving Large, Irregular Graph Problems in X10

IBM Research

Abstract

X10 work scheduling

1. Introduction to X10

X10 is a new Partitioned Global Address Space (PGAS) language being developed at IBM as part of the DARPA HPCS project [**?**]. It is designed to address both programmer productivity and parallel performance for modern architectures from the multicores, to the heterogeneous accelerators (as in the Cell processor), and to the scale-out clusters of SMPs such as Blue Gene. The language is based on sequential Java with extensions for programming fine-grained and massive parallelism. Unlike other PGAS languages such as Co-Array Fortran, Titanium, and UPC whose model of parallelism is Single Program Multiple Data (SPMD), X10 supports dynamic and structured concurrency where SPMD is only a special case. In this section, we provide a brief introduction to the basic concepts in the X10 programming model and the language constructs used to implement the graphs algorithms of interest. For more details and other features of X10, readers please refer to [**?**].

1. **Activities** – All concurrency in X10 is expressed as asynchronous *activities*. An activity is a lightweight thread of execution, which can be spawned recursively in a fork-join manner. The syntax of spawning an activity is `async S`, where a new child activity is created executing statement `S`. Activities can not be named and neither aborted nor canceled. The granularity of an activities is arbitrary - `S` can be a single statement reading a remote variable or a sequence of statements performing a stencil operation on a grid. Our experience has been that this single notion of an asynchronous activity can subsume many levels of parallelism that a programmer may encounter such as threads, structured parallelism (including OpenMP), messaging (including MPI), and DMA transfers.
2. **Places** – The main program starts as single activity at *place* 0. Place is an X10 concept which can be considered as a virtual SMP, but multiple places can be mapped to one physical SMP node. The global address space is partitioned across places. Data and activities have affinity with and only with one place. Activities can only operate on data local to them, that is, within the same place. To access data at another place, a new activity has to be spawned there to perform the operation. The syntax for spawning an activity at place p is `async (p) S`. The diagram in Figure 1 describes the X0 programming model.

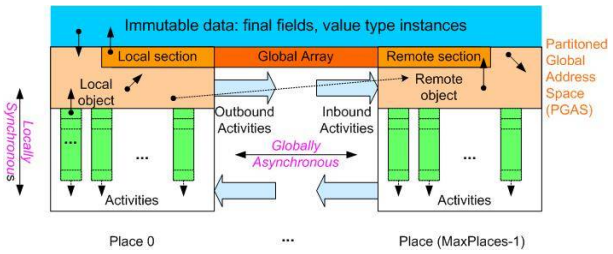

Figure 1. Dynamic parallelism with a Partitioned Global Address Space. All concurrency is expressed as asynchronous activities. Each vertical green rectangle above represents the stack for a single activity. An activity may hold references to remote objects, that is, at a different place. However, if it attempts to operate on a remote object, then it has to spawn a new activity at the remote place to perform the operation. Immutable (read only) data is special which can be accessed freely from any place providing opportunity for single-assignment parallelism.

3. **X10 arrays** – X10 supports a rich set of multidimensional array abstractions and domain calculus as in Titanium [**?**]. The index space is global where each index is an integer vector named *point*, and a *domain* is a set of points which can be either rectangular or not. The *distribution* of an array across places is specified by a `dist`. Each distribution maps a set of points in a region to a set of places. For example, the following code defines a 1D block-distributed array initialized in a way such that each element contains its own

index.

```
region R = [0:TableSize-1];
dist RD = dist.factory.block(R);
long [.] Table = new long[DD]
         (point [i]) {return i;};
```

4. **Parallel loops** – There are two kinds of parallel loop in X10: `foreach` and `ateach`, for looping over a region and a distribution respectively. Their difference is shown by the following two pairs of equivalent statements.

```
foreach (point [i] : R) Table[i]+=i;
   is equivalent to
for (point [i] : R) async Table[i]+=i;
   and
ateach (point [i] : RD) Table[i]+=i;
   is equvalent to
for (point [i] : R)
   async (RD[i]) Table[i]+=i;
```

5. **Finish and clock** – The statement `async S` returns immediately when it is executed even if the statement `S` is not finished, which may also spawn other activities. To wait until a statement `S` has finished globally, that is, all transitively spawned child `async`s have finished, one needs to use the *finish* clause: `finish S`. For example,

```
finish foreach (point [i] : R) Table[i]*=i;
Table[0]+=1;.
```

   Here, after the above two statements are executed, the value of `Table[0]` is guaranteed to be 1. Activities can be synchronized using `finish` by checking their global termination. However, there are many cases in which a barrier-like coordination is needed for a set of activities during the middle of their computation. X10 uses *clock* to coordinate such as set of activities. A clock has phases, and the activities registered with this clock can be synchronized by waiting for their finish of the current clock phase. An activity can be registered with multiple clocks and it can drop any of them at any time.

6. **Atomic blocks** – X10 uses atomic blocks for mutual exclusion. An atomic statement/method is conceptually executed in a single step, while other activities are suspended. An atomic block must be nonblocking, sequential (without spawning activities), and local (no remote data access). Without using the atomic block, the following code will generate race conditions.

```
double sum = 0;
finish foreach (point [i] : R) sum+=1;
```

   Conditional atomic block is another parallel language construct of X10 which can be used, for example, to implement point-to-point synchronization. The syntax for a conditional atomic block is `when (E) S`, where the executing activity suspends until the boolean expression `E` is true, then `S` is executed atomically.

The following example shows how to compute a spanning tree of a connected graph in parallel at one place. A node is in the spanning tree if its field `inTree` is `true`.

```
public class V{
  final int index;
  V parent;
  int degree;
  V[] neighbors;
  boolean inTree=false;
  V(int i){index=i;}
  atomic boolean compareAndSet(){
    if (!inTree) {
       inTree=true;
       return true;}
    else
       return false;
  }
  ...
  public void compute(){
    V node = this;
    for (int k=0; k < node.degree; k++) {
        final V v = node.neighbors[k];
        if (v.compareAndSet(){
          v.parent=node;
          async v.compute();
        }
    }
  }
}
```