Programming Scientific Computations in X10

Tong Wen, Vijay Saraswat, and Vivek Sarkar

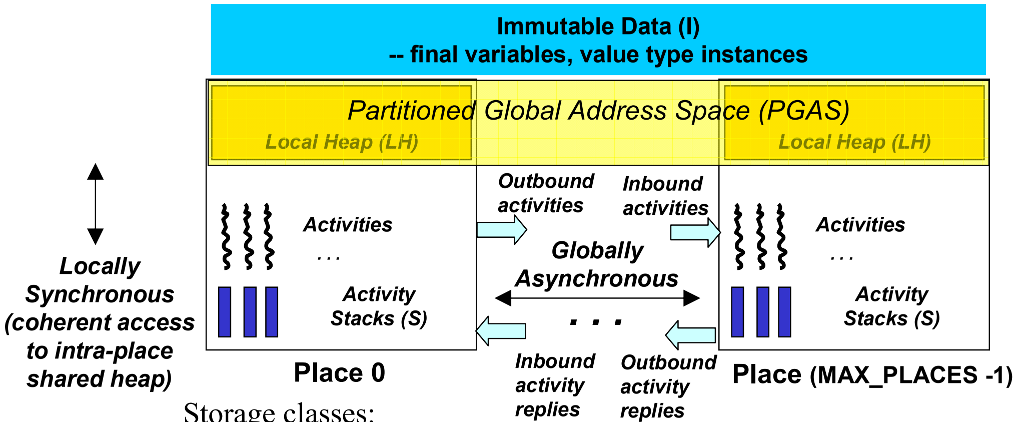
IBM Research

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

Overview

- **X10 is designed for general large-scale and heterogeneous parallel programming**
  - Based on Java with extension to support fine-grained parallelism (data and task parallelism)
  - Not SPMD!
- **Focus on features relevant for programming scientific computing applications**
  - Abstractions for multi-dimensional arrays
  - Serial and parallel looping constructs
  - Constructs for task parallelism
- **Examples: NPB CG and MG benchmarks, LU Factorization, etc.**
  - Build distributed data structures
  - Synchronize concurrent computations
  - Exploit the fine-grained parallelism supported by X10
  - Issues regarding how to handle the hierarchical and heterogeneous nature of the emerging large-scale computer platforms will be addressed in later studies.
- **New features under consideration**

X10 Programming Model



Storage classes:

- Immutable Data (I)
- PGAS
  - Local Heap (LH)
  - Remote Heap (RH)
- Activity Stacks (S)

- *Place* = collection of activities & objects
  - Activities and data objects do not move after being created
- *Scalar object, O* -- maps to a single place specified by O.location
- *Array object, A* – may be local to a place or distributed across multiple places, as specified by A.distribution
- Locality rule: direct access via a remote heap reference is not permitted

X10 Reference Implementation (Operational since 2/2005)

Foo.x10

X10 source program --- must contain a class named Foo with a "public static void main(String[] args) method

x10c Foo.x10

x10c

X10 compiler --- translates Foo.x10 to Foo.java, uses javac to generate Foo.class from Foo.java

Foo.class

Foo.java

X10 program translated into Java --- // #line pseudocomment in Foo.java specifies source line mapping in Foo.x10

x10 Foo

X10 Virtual Machine (JVM + J2SE libraries + X10 libraries + X10 Multithreaded Runtime)

X10 extern interface

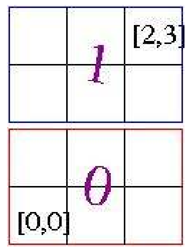
External DLL's

X10 Abstract Performance Metrics

X10 Program Output

Abstractions for Multi-dimensional Arrays

- **Point, Region,** and **Distribution**
  - **Point**: a vector of integers
  - **Region**: a (rectangular) set of Points
  - **Distribution**: a map from a Region to a set of Places
- **X10 arrays**: defined on Distribution




```
point p=[0,1,2,3];

region r1=[0:2, p[0]:p[1]];
region r2=[0:2, p[2]:p[3]];

dist d1=(r1->place.factory.place(0));
dist d2=(r2->place.factory.place(1));

final double [.] A_dist=new double
    [d1||d2];
final double [.] A_local=new double
    [r1||r2];
```

```
final double [.] A_local=new double [(r1||r2)->here];
```


Serial and Parallel Looping Constructs

- Serial loop:
  - **for (point p: r1) {…}**
  - **for (point p[i,j]: r1) {…}**
- Parallel loop:
  - **foreach (point p: r2) {…}**
  - **foreach (point p[i,j]: r2) {…}**
- Distributed parallel loop:
  - **ateach (point p: d1||d2) {…}**
  - **ateach (point p[i,j]: d1||d2) {…}**

```
region R=[-1:256,-1:256], r=[0:255,0:255];

final point North=[0,1], South=[0,-1],
          West=[-1,0], East=[1,0];

final double [.] A=new double [R];
…
for (point p: r)
    A[p]=(A[p+North]+A[p+South]+A[p+West]
          +A[p+East]-4*A[p])*h;
/*
foreach (point p: r)
    A[p]=(A[p+North]+A[p+South]+A[p+West]
          +A[p+East]-4*A[p])*h;
*/
```

Constructs for Task Parallelism

- Spawning an asynchronous activity:
  - **async (P) S**
    - Create a new activity to execute statement S at place P
  - **future (P) {E}**
    - Create a new activity to evaluate expression E at place P
- Converting local termination to global termination:
  - **finish S**
    - Execute S as usual, but wait until all activities spawned (transitively) by S have terminated before completing the execution of finish S
  - **future (P) {E}.force()**
    - Caller blocks until return value is obtained from future (and all activities spawned transitively by E have terminated)
- Clocks and (conditional and unconditional) atomic blocks
  - Synchronization

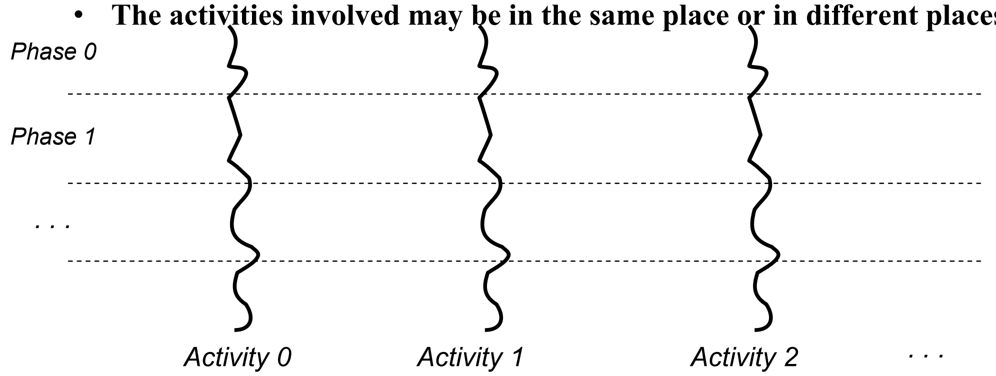
finish

```
foreach (point p: r)
    A[p]=(A[p+North]+A[p+South]+A[p+West]
            +A[p+East]-4*A[p])*h;

/*
for (point p: r) async ( here )
    A[p]=(A[p+North]+A[p+South]+A[p+West]
            +A[p+East]-4*A[p])*h;
*/

final dist D=dist.factory.unique();
ateach (point p: D)
    System.out.println (here.toString());

/*
for (point p:D) async ( D[p] )
    System.out.println (here.toString());
*/
```

More examples to come!

X10 clocks: Motivation

- Activity coordination using finish and force() is accomplished by checking for activity termination
- However, there are many cases in which a producer-consumer relationship exists among the activities, and a “barrier”-like coordination is needed without waiting for activity termination
- **Clock** is a generalization of barriers
  - **The activities involved may be in the same place or in different places**



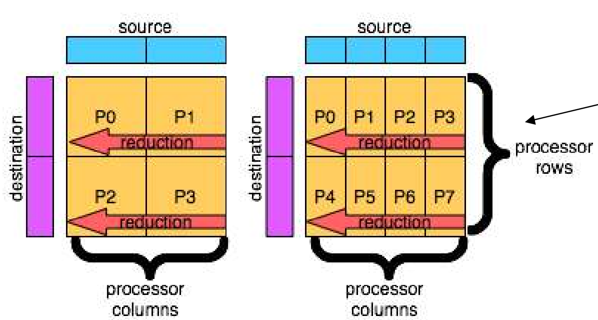
Atomic Blocks

- **Unconditional atomic block:**
  - **atomic S**, where **S** is a statement
  - The above statement is executed by an activity as if in a single step, during which all other concurrent activities in the same place are suspended.
- **Conditional atomic block:**
  - **when (C) S**, where **C** is a Boolean expression
  - The above statement is executed atomically when **C** becomes true.
  - In our examples, point-to-point synchronization is implemented using conditional atomic blocks.
  - Using conditional blocks can lead to deadlock!

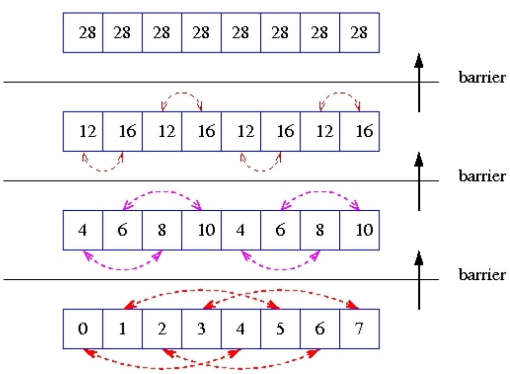
NPB CG

- **Use inverse power method to compute an approximation to the smallest eigenvalue of sparse symmetric positive definite matrix**

```
public static Vector cg(SparseMat A, Vector x, Vector r){
    double d, rho;
    double alpha, rho0,beta;
    Vector p=new Vector(A.m_N);
    Vector q=new Vector(A.m_N);
    Vector z=new Vector(A.m_N);

    r.copy(x); p.copy(x); rho=r.dot(r);

    for (int i=0;i<10;i++){
        A.multiply(q,p);
        d=p.dot(q);
        alpha=rho/d;
        z.axpy(alpha,p,z);
        rho0=rho;
        r.axpy(-alpha,q,r);
        rho=r.dot(r);
        beta=rho/rho0;
        p.axpy(beta,p,r);
    }
    return z;
}
```



All Reduction (Barrier) in Matrix-Vector Multiplication (CG)

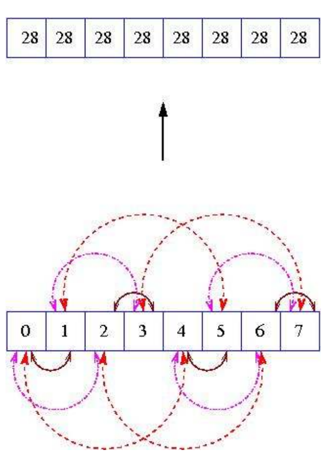



```
final dist ALLPLACES=dist.factory.unique();
final double [.] A=new double [ALLPLACES] (point[i]){return i;};
final double [.] B=new double [ALLPLACES];
final int factor=place.MAX_PLACES;
final int phases=(int)(Math.log(factor)/Math.log(2));
finish async{
        final clock clk=clock.factory.clock();
        ateach (point [i]:ALLPLACES) clocked(clk){
                boolean red=true;
                int Factor=factor;
                int shift;
                for (int j=0;j<phases;j++){
                        shift=Factor/2;
                        final int
                        destProcID=(i+shift)%Factor
                                        +i/Factor*Factor;
                        if (red){
                                B[i]=future(A.distribution[destProcID])
                                        {A[destProcID]}.force();
                                B[i]+=A[i];
                        }
                        else{
                                A[i]=future(ALLPLACES[destProcID])
                                        {B[destProcID]}.force();
                                A[i]+=B[i];
                        }
                        next;
                        Factor/=2; red=!red;
                }
                if (!red) A[i]=B[i];
        }
}
```

B[i]=A[destProcID]

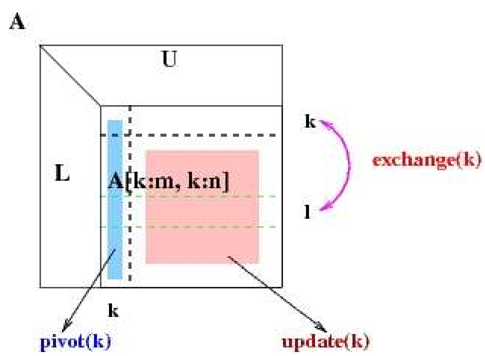
A[i]=B[destProcID]

All Reduction (Point-to-Point) in Matrix-Vector Multiplication (CG)




```
final dist ALLPLACES=dist.factory.unique;
final double [.] A=new double [ALLPLACES] (point[i]){return i;};
final double [.] B=new double [ALLPLACES];
final int [.] Flag1=new int [ALLPLACES] (point [i]){return -1;};
final int [.] Flag2=new int [ALLPLACES] (point [i]){return -1;};
final int factor=place.MAX_PLACES;
final int phases=(int)(Math.log(factor)/Math.log(2));

finish  ateach (point [i]:ALLPLACES){
   int Factor=factor;
   int shift;
   for (int j=0;j<phases;j++){
           shift=Factor/2;
           final int destProcID=(i+shift)%Factor+i/Factor*Factor;
           when (Flag1[destProcID]==(j-1)) {}
                   B[i]=A[destProcID];
                   async (ALLPLACES[destProcID]) atomic
                           Flag2[destProcID]++;
           when (Flag2[i]==j) {}
                   A[i]+=B[i];
                   Flag1[i]++;

           Factor/=2;
   }
}
```

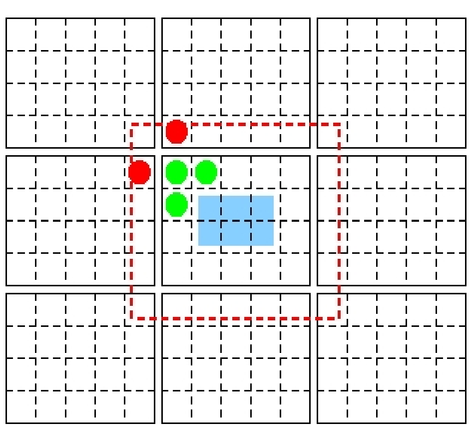
LU Factorization




```
public void lu(final double [.] a_A){ //the matrix is assumed to have any distribution
        assert a_A.rank==2;
        m_A=a_A;
        m_R=a_A.region;
        m_D=a_A.distribution;
        m=m_R.rank(0).size(); n=m_R.rank(1).size();
        final int steps=Math.min(m,n)-1;
        final region STEPS=[0:steps-1];
        m_p=new int [steps+1] (point [i]) {m_p[i]=i;};

        /* flags for synchronization*/
        m_rowScore=new int [[0:m-1]] (point [i]) {return m;};
        m_update[0]=true; m_update[1]=true;

        double res;
        for (point [k]:STEPS){
           when (m_update[0]) {}
                res=pivot(k);
           if (res!=0){
                final int score=n-k;
                when ((m_rowScore[k]==score) && (m_rowScore[m_pivotInfo]==score)){}
                    exchange(k);
                when (m_update[1]){m_update[0]=false;m_update[1]=false;}
                    async update(k);
           }
           else{
                when (m_update[1]){};
           }
        }
}
```


NPB MG

- Use multigrid method to solve Poisson’s equation (3D)



- Construct the distributed array

```
value Wrapper{
            double [.] m_array;
            Wrapper(final double [.] a_array){
                       m_array=a_array;
            }
            …
}
…
region [ ] m_REGIONs;
dist [] m_DIST;
Wrapper value [.] m_u;
…
for (point p: m_placeGrid){
…
      m_DIST[i]=m_REGIONs[i]->place.factory.place(i);
      m_u[i]=new Wrapper(new double [m_DIST[i]]);
…
}
```

Overlapping Communications and Computations in MG

- **Bulk-synchronous style**:
  - Update ghost values first
  - Perform local stencil operations in parallel

```
finish updateGhost();
finish ateach(point [i]: AllPLACES){
        for (point p:localRegion(i))
                stencilOp(p);
}
```

- **Overlapping communication with computation**:

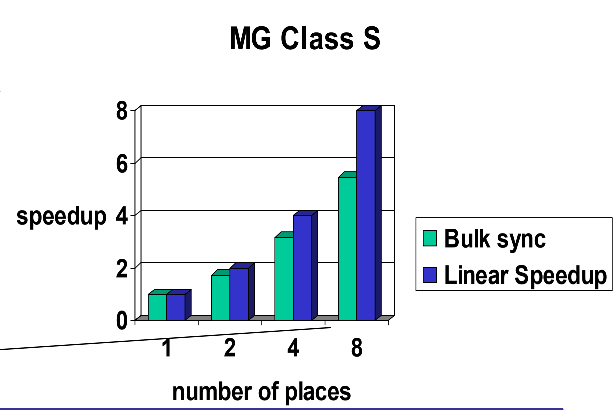
```
finish {
  //Inner region
  ateach(point [i]:ALLPLACES){
      for (point p: InnerLocalRegion(i))
          stencilOp(p);
  }

  // Ghost layer
  finish updateGhost();

  //Boundary layer
  ateach(point [i]:ALLPLACES){
      for (point p: LocalRegion(i)-InnerLocalRegion(i))
          stencilOp(p);
  }
}
```


An Abstract Performance Model (Simulation)

- **Abstract system parameters**:
  - FP results per clock
  - Communication latency
  - Communication Bandwidth
- **Computation cost:**
  - Flops/FP_PERCLOCK
- **Inter-place communication cost**:
  - LATENCY + Doubles*8/BANDWIDTH
- **Costs are inserted into code manually and the runtime computes the length of critical path**
- **Ideal speedup as the measure of how much parallelism that can be expressed in X10**

Comp cost: 3,272,372
CPL: 599,912
% .
= **Ideal** speedup: 5.46

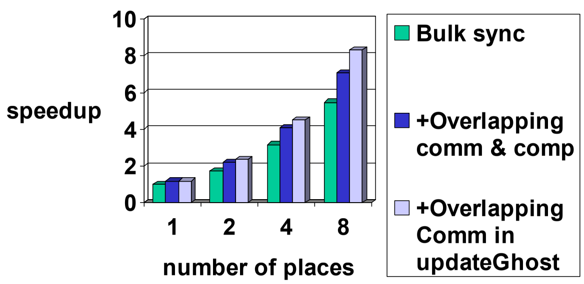
- **A case where comm and comp ratio is high:**
  - FP_PERCLOCK=4
  - LATENCY=375 (cycles)
  - BANDWIDTH=5.3 (bytes/cycle)
  - The small test problem: 32x32x32 (Class S)
    Computation cost: 3,272,372 (cycles)



Multiple Levels of Parallelism

- **Overlapping communication with computation in stencil operations in MG**
- **Overlapping communications in updating the ghost values in MG**
  – Three phases one for each dimension (3D grid)
  – At each phase, the two faces of ghost values can be updated simultaneously

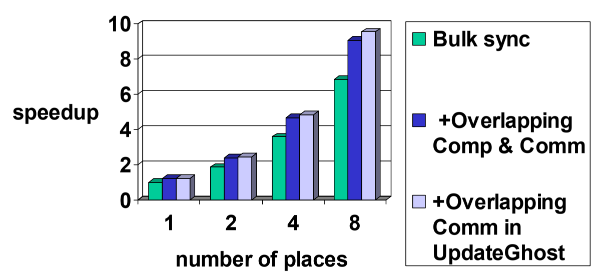
- **Fine-grained parallelism**
  – Replacing unordered for loops with foreach loops
  – For MG, the stencil operations can speed up by a factor of 5382 (at most)
  – The speedup factors (1,2,4,8)
    5382.2, 27.6, 32.8, 37.3




```
finish ateach (point [i]: AllPLACES){
        foreach (point p:localRegion(i)){
            stencilOp(p);
}
}
```


More Abstract Performance Metrics

- **Smaller latency:**
  – FP_PERCLOCK=4
  – LATENCY=37.5 cycles
  – BANDWIDTH=5.3 bytes/cycle
  – The small test problem: 32x32x32 (Class S)
    Computation cost: 3,272,372 cycles
- **Latency is the bottleneck of scalability**



Latency	Speedup			
	1	2	4	8
375	5382.2	27.6	32.8	37.3
37.5	5382.2	53.0	77.3	112.3

Mapping Fine-Grained Parallelism onto Coarse-Grained Machine

- **Explicitly:**
  - Instead of having one activity per point, one can spawn an activity for a sub region
    - Hierarchical tiled region
    - Hierarchical tiled array
- **Implicitly:**
  - Compilation and runtime technologies to optimize the foreach loop
  - Programmers express the logical parallelism in their code and let compiler and runtime to handle the mapping from logical parallelism to hardware threads.

Improvements and New Features

- **Expressing logical parallelism in X10 is easy. The challenge is how to map fine-grained parallelism to coarse-grained machine.**
- **Improvements and new features**
  - Arrays:
    - Array views
    - Array copy method
    - More dimension independent syntax
    - HTR and HTA
    - Automatic support for ghost values
  - Dependent types
  - Implicit syntax
    - B[i]=A[destProcID]
    - B[i]=future(A.distribution[destProcID]){A[destProcID]}.force();
    - Add implicit finish to **foreach (point: R) S**
      finish for (point:R) async S
    - Add implicit finish to **ateach (point: D) S**
      finish for (point:R) async (D[p]) S
  - Multi-phase clock ?

Acknowledgement
- Kaushik Datta. Titanium Group, UC Berkeley.

X10 Applications/Benchmarks

- **Java Grande Forum**
  - OOPSLA Onwards! 2005
  - Showed substantial (SLOC) benefit in serial → parallel → distributed transition for X10 vs Java (qua C-like language).
- **SSCA**
  - SSCA#1 (PSC study)
  - SSCA#2 (Bader et al, UNM/GT)
  - SSCA#3 (Rabbah, MIT)
- **Sweep3d**
  - Jim Browne

- **NAS PB**
  - CG, MG (IBM)
  - CG, FT, EP (Padua et al, UIUC)
  - Cannon, LU variant (UIUC)
- **AMR** (port from Titanium)
  - In progress, IBM
- **SpecJBB**
  - Purdue