X10:
An Object-Oriented Approach to Non-Uniform Clustered Computing

秋山 茂樹

全体ミーティング (2010/7/27)

X10: An Object-Oriented Approach to Non-Uniform Clustered Computing

- 並列プログラミング言語 X10 の概要について述べた論文
  - 著者
    - IBM T.J. Watson Research Center
    - University of California, Los Angeles
    - IBM Toronto Laboratory
  - 発表場所
    - OOPSLA'05

X10

- HPC 向けの並列プログラミング言語
  - ノードが NUMA であるクラスタ (NUCC) を想定
  - IBM PERCS プロジェクトの一環
    - 2010 年までに並列アプリの生産性を10倍に
  - まだ発展途上
    - 2004年時点: ver. 0.4.1, 2010年現在: ver. 2.0.5

Motivation

- これまでの並列分散機構は NUCC に適さない
  - Thread, java.util.concurrent, java.rmi
    - ノード内での不均一なメモリアクセスを表現できない
  - MPI
    - ソフトウェアの複雑化
    - システムに詳しい人でも難しい
- NUCC システムのための新しい言語が必要

Goals (1/2)

- 安全性
  - エラーを起こしにくいような言語設計
  - 静的型チェックによる検証
- 解析のしやすさ
  - コンパイラ、静的解析、リファクタリング
  - 機能のシンプルさと一般性が必要

Goals (2/2)

- スケーラビリティ
  - スケーラブルでない機能を入れない
- 柔軟性
  - 性能向上のためには種々の並列性利用が不可欠
    - データ並列、タスク並列、パイプライン化、etc...
    - SPMD モデルは不適

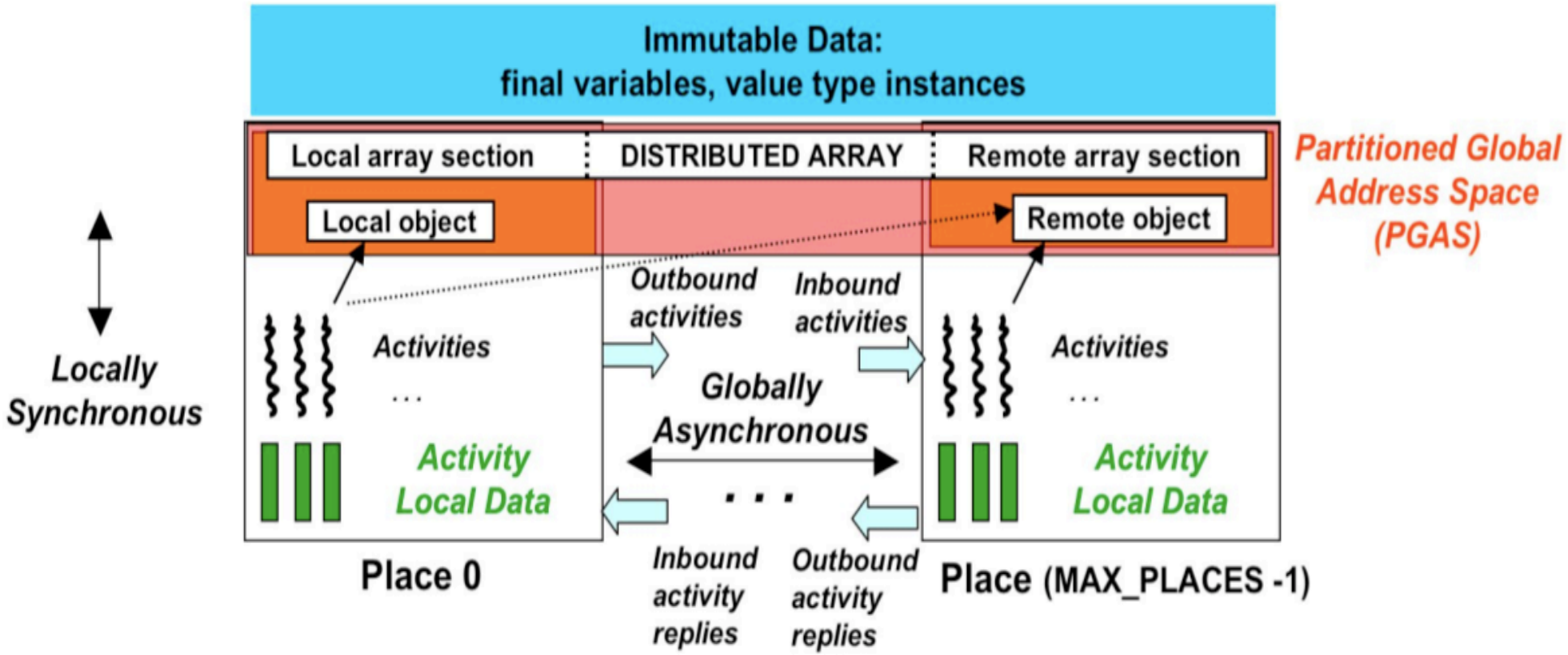
X10 逐次サブセット

- Java like な言語に加えて
  - Value classes
  - Region iterators
  - Type inference (略)
  - Constraint types (略)
  - etc...

X10 Programming Model

- lightweight threads
  - Asynchronous Activities, Foreach, Future
  - Atomic Blocks
- Partitioned Global Address Space (PGAS)
  - Places, At, Distributions, Ateach
  - Asynchronous Expression
- Clocks
- Value Classes

X10 Programming Model



Place

- (大雑把に言うと) 一つのノードを表現
  - オジェクトとスレッド(Activity)の集合
  - ファーストクラスオブジェクトとして扱える
    - `here` で現在の place を取得
    - `obj.home` でオブジェクトの属する place を取得
    - `here.next()`, `here.prev()` など
  - CUDA Core や Cell の SPE を扱うことも想定 (?)

Place の性質

- Place の増減、ノード間での移動は不可 *1
- Place 間の mutable object, activity の移動は不可
  - immutable object は可
  - 他の place での activity の生成は可
- Place は仮想的
  - 計算資源への割り当てはランタイム次第

*1 place の hierarchical, dynamically varying notion は future work

Asynchronous Activities

- activity は lightweight thread
  - 生成: `async (P) S` cf. Cilk's spawn
    - ステートメント `S` を実行するスレッドを生成
    - `P` は activity を生成する place
  - 同期: `finish S` cf. Cilk's sync
    - `S` 内で起動した activity が終了するまで待つ

Asynchronous Activities: 例

fibonacci

```
class Fib {
  var r:int;
  public def this(n:int) {
      this.r = n;
  }
  public def run() {
    if (r < 2) return;
    val f1 = new Fib(r-1);
    val f2 = new Fib(r-2);
    finish {
      async f1.run();
      f2.run();
    }
    r = f1.result
      + f2.result;
  }
}
```

data copy

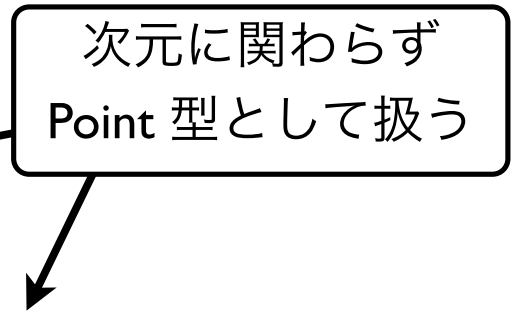
```
val t1 = new T();
finish async (here.next()) {
  val t2 = new T();
  async (t1) {
    t1.value = t2;
  }
}
```

Partitioned Global Address Space

- Globally Asynchronous, locally Synchronous (GALS)
  - local のデータにはアクセスできるが remote のデータにはアクセスできない (*Locality Rule*)
    - remote のデータにアクセスする際には、その place に activityを生成し、アクセスする
  - place 内では sequential consistency が成り立つ (?)
    - 実装が保証

Arrays, Regions, Distributions

- Region
  - (多次元)配列の index (point) の集合:
    - `[0..9]`: 集合 { (0), (1), ..., (9) }
    - `[0..9, 0..9]`: 集合 { (0, 0), (0, 1), ..., (1, 0), ... (1, 1) }
  - Region に対する操作
    - union, intersection, difference, etc...



Arrays, Regions, Distributions

- Distribution
  - region のどの部分をどの place に割り当てるか
  - 組み込みの distribution
    - unique distribution
    - constant distribution
    - block distribution
    - block-cyclic distribution
    - etc...

Arrays, Regions, Distributions

- Distributed Array
  - 複数の Place にまたがる(多次元)配列
  - 生成する際に distribution を指定

二乗和の計算

```
val a = DistArray.make[Int](
              [1..10]->here,
              ((i):Point) => i);
val sumsq = a.map((n:int) => n*n).reduce(Int.+, 0);
```

Foreach, Ateach

- 並列ループ: `foreach (p in C) S`
  - C に含まれるすべての p について、
    p.home で S を実行

配列のコピー

```
val a = Array.make[Int](((i):Point) => i);
finish foreach ((i) in a.region) {
    b(i) = a(i);
}
```

Foreach, Ateach

- 並列ループ: `ateach (p in D) S`
  - distribution `D` に含まれるすべての point `p` について対応付けられた place で `S` を実行

配列のコピー

```
val a = DistArray.make[Int](dist, ((i):Point) => i);
finish ateach ((i) in a.dist) {
    a(i) = b(i);    // assert(a.dist == b.dist)
}
```

Clocks

- バリア機構
  - activity に登録: `async clocked (c) S`
  - バリア: `next`

```
finish async {
  val c = Clock.make();
  async clocked (c) {
    say(“A-1”); next; say(“A-2”); next; say(“A-3”);
  }
  async clocked (c) {
    say(“B-1”); next; say(“B-2”); next; say(“B-3”);
  }
}
```

Atomic Blocks

- アトミックブロック: `atomic S`
  - ステートメント `S` の実行をアトミックに行う
  - 制約: `S` には async や finish に類する操作を書けない

```
var sum = 0;
finish foreach ((i) in [1..9]) {
  atomic {
    sum += i;
  }
}
System.OUT.println(i);  // 45
```

Future

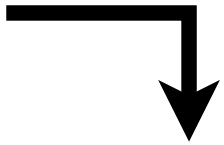
- Future: `val F = future (P) E;`
  - place `P` で式 `E` を非同期実行
  - `F.force()` で計算が終わるのを待ち、結果を取得

```
def fib(n:Int) {
  if (n < 2) return n;
  val f1 = future fib(n-1);
  val f2 = future fib(n-2);
  return f1.force() + f2.force();
}
```

At Expression

- 同期実行: `val V = at (P) E;`
  - place `P` で式 `E` を実行し、実行が終わるのを待ってその結果を返す

```
val t1 = new T();
finish async (here.next()) {
  val t2 = new T();
  async (t1) {
    t1.value = t2;
  }
}
```



```
val t1 = new T();
t1.value
  = at (here.next()) new T();
```

Value Classes

- 変更可能なフィールドを持たないクラス
  - value class のインスタンスはどの place からでもアクセスできる
    - `at (P) S` を使用する必要はない
  - built-in value classes
    - boolean, byte, char, int, String, etc...

Property

- async, finish, at, atomic, clock を用いて記述されたプログラムはデッドロックしない
  - wait が発生するのは finish, at, clock

wait-for グラフ: 各 activity における wait の依存関係を表現したグラフ

A1 → A2 なら A1 は A2 の終了を待っている
A1 → C → A2 なら A1 は A2 の next を待っている

1. suspend した activity と clock をノードとしたグラフを考える
2. 各 clock からその clock が登録された activity(finish) へエッジを引く
3. 各 activity(next) からその activity に登録された clock へエッジを引く
4. 各 activity(finish S) から S で spawn された各 activity へエッジを引く

このとき、このグラフにサイクルができていれば deadlock