Slice, Partition and Reforest for Data Access and Distribution

William Cook
with
Eli Tilevich, Yang Jiao, Virginia Tech
Ali Ibrahim, Ben Wiedermann, UT Austin

IFIP WG 2.11 April, 2009

Let's reinvent Remote Procedure Calls

1. Clean interfaces
   - Don't have to design differently for distribution
2. Latency
   - As few communications as possible

     3 to 10 round-trips per second
   - Clear performance model
3. Simple memory model
   - What about remote pointers?
4. Control partial failures
5. Stateless servers for scalability
6. Simple programming model
   - Compositional

Clean Service Interface

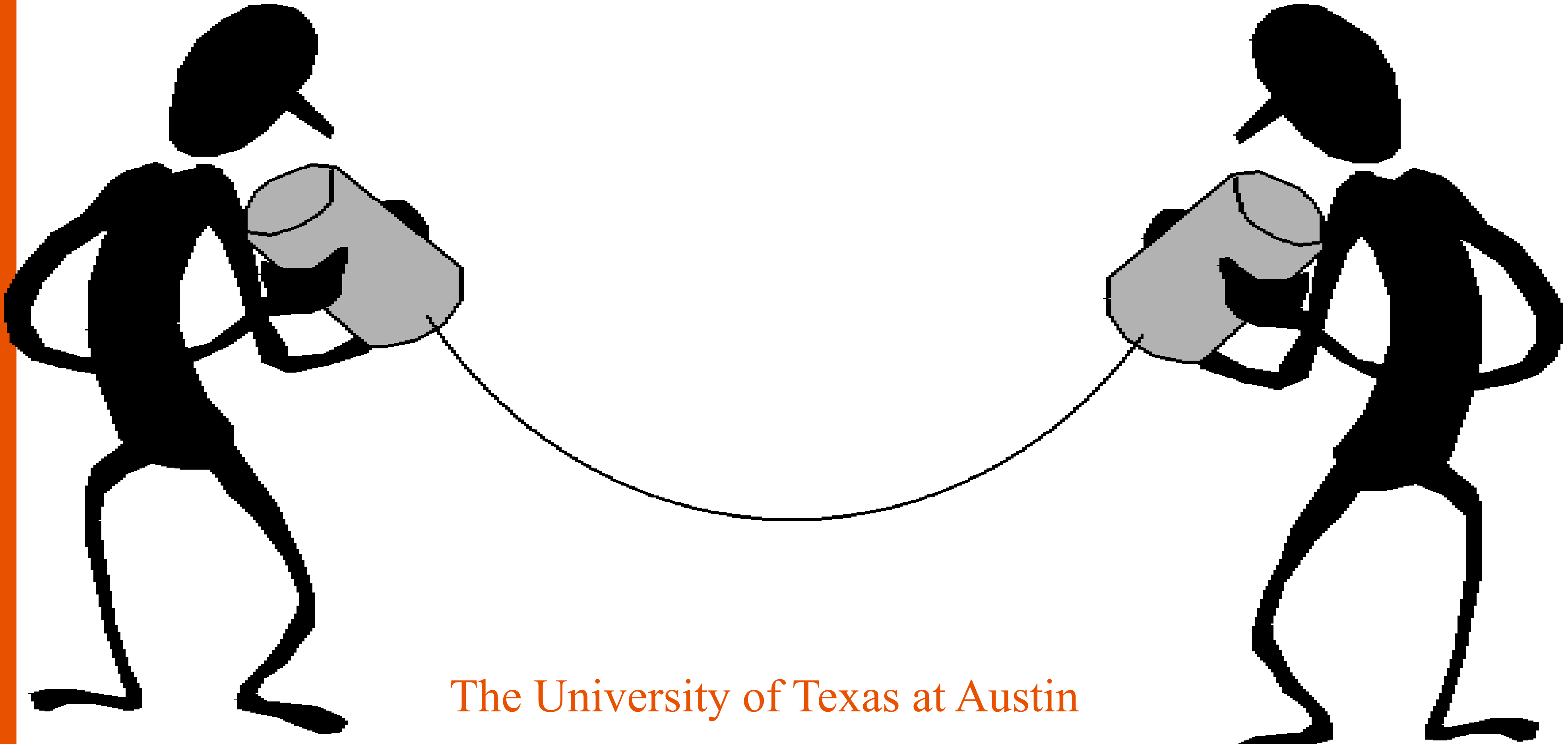
```
interface Music {
  Album[] getAlbums();
  Album getAlbum(String name);
}

interface Album {
  String getTitle();
  void   play();
  int    rating();
  void   merge(Album other);
}
```

We could also use ML-style module to define remote service interface

Latency?

- Simple procedure call

```
print( album.getTitle() );
```



Latency?

- Simple procedure call

```
print( album.getTitle() );
```

- But what about multiple calls?

```
print( album.getTitle() );
print( album.rating() );
```

- RPC model gives two round trips
  - Can we do this in one round trip?
  - Alternative is *asynchronous* calls... more later

Remote Batch Invocation (RBI)

- New statement: batch block

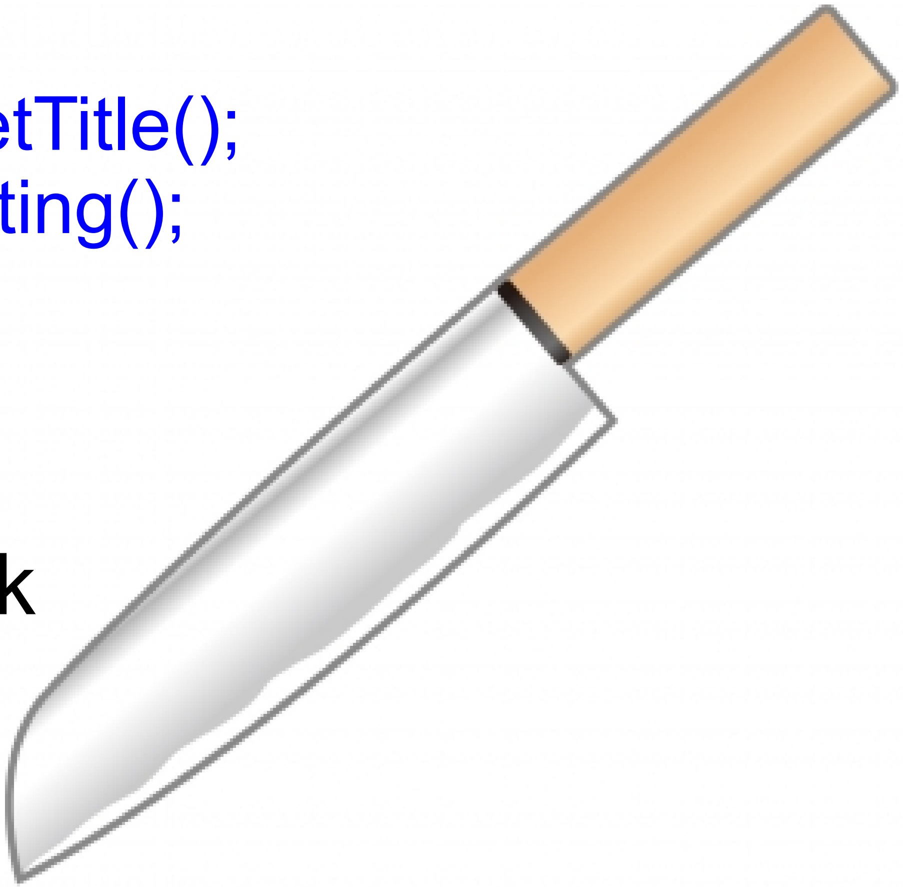
```
batch (album) {  // album is service root
    print( @album.getTitle() );
    print( @album.rating() );
}
```

- Semantics: @remote parts executed first
- Clear performance model
  - Executes all remote actions in one round-trip
- Simple programming model
  - Reduces partial failures

Partition

- Partition batch into two parts, remote and local:

  remote computation:
  ```
      data.p1 = album.getTitle();
      data.p2 = album.rating();
  ```
  local computation:
  ```
      print( data.p1 );
      print( data.p2 );
  ```
- Data is transfered in bulk
- Related to
  - *remote evaluation*
  - *binding time analysis (binding location analysis?)*



What else can go in a batch?

Asynchrony does not help!

- Composition

```
batch (r) { @r.foo().bar().getName(); }
```

- Conditions

```
batch (a) { if (@a.rating() > 50) @a.play(); }
```

- Loops

```
batch (music) {
   for (Album a : @music.getAlbums() )
    if (a.rating() > 50)
      print( @a.getName() + ": " + @a.rating() );
}
```

- Exceptions work too

Partitioning Loops

- Partition batch into two parts, remote and local:

```
remote computation:
   for (Album a : music.getAlbums() ) {
      item = data.add();
      item.p1 = a.getName();
      item.p2 = a.rating() );
   }
// local computation
   for ( item : data.iterations() )
       print( item.p1 + ": " + item.p2 );
```

- Data is a list of pairs
- Runs the loop twice (same for conditions)

What about the data?

A new idea:

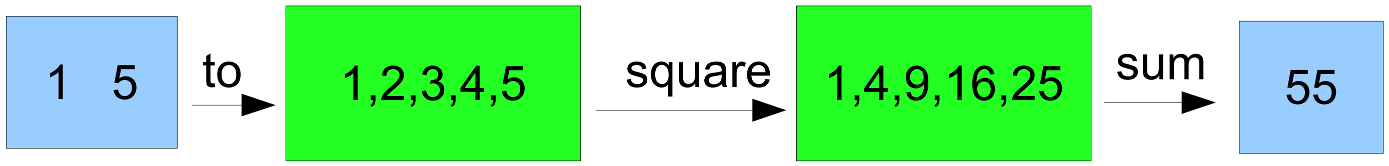
Reforestation

Introduce intermediate data structures

Deforestation [Wadler 89]

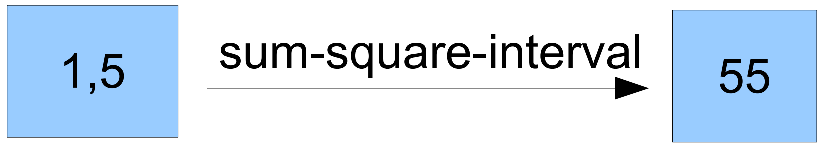
- Remove intermediate data structures (trees)

  sum (square (1 `to` 5))



- Deforested version

  sum-square-interval(1, 5)



Reforestation

- Split program P(r) in two:

  $P(r) = P_2(P_1(r))$

  or

  $P(r) = \textbf{let}\ data = P_1(r)\ \ \textbf{in}\ \ P_2(data)$

- Adding intermediate structure is efficient because of remoteness

Memory Model

- Only transfer primitive values!
- No proxies (remote pointers)
  - Server is stateless, “service oriented”
  - No distributed garbage collection
- Serialization by public interfaces

```
batch (remote) {
    RemoteSet r = @remote.makeSet();
    for (int elem : localSet().items() )
        @r.add( elem );
    ....
```

  - Illegal: RemoteSet r = localSet;
  - Need reusable helper functions/coercions

Evaluation

	RMI CORBA	Web Services	Remote Batch Invocation
Clean Interfaces	Good	Bad	Good
Latency	Bad	Good	Good
Memory model	Bad	Good	Good
Stateless	No	Yes	Yes
Partial Failure	Bad	Better	Better
Programming Model	Good	Bad	Good... but...

Re-ordering

- Statements are reordered! @'s run first

```
batch (remote) {
    local.update( @remote.get() );
    @remote.set( local.get() );
}
```

- Partitions to:

```
remote execution: {
    data = remote.get();
    remote.set( local.get() );  // local.get() happens first!
}
// local execution
local.update( data );
```

Generalized Batches

- Parameterize by batch handler
  - **batch** RMI (remoteObject) { … }
  - **batch** WebService (service) { … }
  - **batch** SQL (db) { … }
  - **batch** GPU (gpu) { …. }
  - **batch** PartialEval (s) { … }
  - **batch** H (r) B = B_2(H <B_1>(r))
- Batch provides generalized program partitioning and reforestation capability

Web Services: Document = Batch Amazon Web Service

```
<ItemLookup>
<AWSAccessKeyId>XYZ</AWSAccessKeyId>
<Request>
  <ItemIds>
      <ItemId>1</ItemId>
      <ItemId>2</ItemId>
  </ItemIds>
  <IdType>ASIN</ItemIdType>
  <ResponseGroup>SalesRank</ResponseGroup>
  <ResponseGroup>Images</ResponseGroup>
</Request>
</ItemLookup>
```

```
interface Amazon {
  void login(String awsKey);
  Item getItem(String ASIN);
  ...
}
interface Item {
  int getSalesRank();
  Image getSmallImage();
  ...
}
```

```
// calls specified in document
aws.login("XYZ");
Item a = aws.getItem("1");
Item b = aws.getItem("2");
return new Object[] {
  a.getSalesRank(), a.getSmallImage(),
  b.getSalesRank(), b.getSmallImage()    }
```

Batching Database Access

```
batch SQL (Database db : dbService) {
  for (Album album : @db.getAlbums())
    if (@(album.rating() > 50))
      System.out.println("Played: " + @album.getTitle());
}
```

```
DbResults data = dbService.executeQuery(
              "select title from albums where rating > 4");
for (item : data.items())
  System.out.println("Played: " + item.getTitle());
```

- Also updates, aggregation and grouping

Maier 1987

"Whatever the database programming model, it must allow complex, data-intensive operations to be *picked out* of programs for execution by the storage manager, rather than forcing a record-at-a-time interface."

Related work

- Automatic program partitioning
- Remote evaluation (mobile code)
- Implicit batching
- Asynchronous remote invocations
- Transactions (batch/atomic)

Contributions

- New statement form:
  - **batch** C (r) { body }
- Interesting semantics, general applications
  - Partition
  - Reforest
- Unifies distribution and data access
  - Can be asynchronous too