Language Overview

- **Orchestration code**
  - Describes global control flow
  - Macro dataflow approach

- **Separate sequential code**
  - Defines local components and methods
  - Standard C/C++ code

- **Translated into Charm++ code**
  - Taking advantage of ARTS benefits
    - Adaptive overlap, automatic load balancing, etc.
Object Arrays

- Collection of objects indexed by a general mechanism
- Array declaration and instantiation
  
  ```
  class Cell : ChareArray2D;
  class CellPair : ChareArray4D;
  
  obj cells :  Cell[N,M];
  obj cellpairs :  CellPair[N,M,N,M];
  ```

- Invoking method on an object
  
  ```
  myMain.foo();
  cells[0,0].foo();
  ```
**foreach Statement**

- Invokes a method across all elements in an array

```
foreach i in myWorkers
    myWorkers[i].doWork(1,100);
end-foreach
```

```
foreach x,y in cells
    cells[x,y].integrate();
end-foreach
```

- Nested foreach statement is meaningless
Input and Output of A Method

- Input and output of a Charisma method

```c
foreach i in workers
    (q[i]) <- workers[i].foo(p[i+1]);
end-foreach
```

- Method `workers::foo` produces the value `q`, and consumes value `p`

- Multiple or none `inports` and `outports`

```c
(q[i]) <- workers[i].foo(p[i-1], p[i], p[i+1]);
```

- Produced value must have same index as object’s “i”

  Consumed value with an index in the form of “i±c”
Parameter Space

- Variables used in inports/outports constitute the “parameter space”
  - Declared and used in orchestration code
  - Type be intrinsic types, user-defined data type or arrays

```plaintext
param error : double;
param atoms : AtomBucket;
param celldata : double [CELLSIZE];
```
Program Order

- Program order is used to determine data dependence
  - An *inport* consumes the value produced by the most closely preceding statement with *outport* on the same variable
  - No implicit barrier between foreach statements

- Control transfer determined by data availability

```plaintext
foreach i in A
    (p[i]) <- A[i].foo();
end-foreach
foreach i in B
    B[i].bar(p[i-1]);
end-foreach
```

Diagram:
```
A[1]   
    |   
    v   
    . .
    |   
    v   
    . .
    |   
    v   
    . .
    |   
    v   
    . .
    |   
    v   
    . .
B[2]   
    |   
    v   
    . .
    |   
    v   
    . .
    |   
    v   
```
Loop Statement

- Data dependence in loops (for and while)
  - First inports in loop body connect with
    - Last outports before loop (for first iteration), and
    - Last outports in the loop body (for following iterations)
  - At the last iteration, the last unconsumed outport values will be consumed by code following the loop

(q[...]) <- ...
for iter = 1 to MAX_ITER
  foreach i in A
    (p[i]) <- A[i].foo(q[i+1]);
  endforeach
  foreach i in A
    (q[i]) <- A[i].bar(p[i-1]);
  endforeach
end-for
...(q[...])

iter = 1
iter = 2, 3, ...
iter = MAX_ITER
Program Determinacy

- Deterministic execution
  - For any individual object, Charisma methods are always executed in the program order

- Enforcing determinacy
  - State counter in object for executing methods in program order
  - Iteration epoch control
    - Avoid sending value to next iteration prematurely
    - Impose barrier where necessary
Sequential Methods

- Consumed values passed in as ordinary parameters
- Produced values indicated by keyword “outport”
- Producing with “produce” and “reduce” keywords

```
(q[i]) <- workers[i].foo(p[i+1]);
```

```
WorkerClass::foo(double p, outport q){
    ... = p;
    double local_q = ...;
    produce(q, local_q);
    ...
}
```
Communication Patterns

Charisma is capable of expressing various communication patterns

- Point-to-point
- Reduction
- Multicast
- Gather
- Scatter
- All-to-all operation
Communication Patterns (1)

- Point-to-point communication

```c
foreach i in A
    (p[i]) <- A[i].f(...);
end-foreach
foreach i in B
    (...) <- B[i].g(p[i]);
end-foreach
```

- Sequential code: producing a scalar

```c
AClass::f(..., outport p){
    produce(p, local_p);
}
```

- Sequential code: producing an data array

```c
AClass::f(..., outport p){
    produce(p, local_p_arr, arr_size);
}
```
Communication Patterns (2)

- **Reduction**: indicated by a “+” sign before the published value

  ```
  foreach i,j in A
    (+err) <- A[i,j].bar(...);
  end-foreach
  Main.test(err);
  ```

- **Sequential code: reduction operator**

  ```
  AClass::bar(..., outport err){
    reduce(error, local_err, CHARISMA_SUM);
  }
  ```
Charisma++ example (Simple)

Jacobi 1D

begin
  forall i in J
    <lb[i],rb[i]> := J[i].init();
  end-forall
  while (e > threshold)
    forall i in J
      <+e, lb[i], rb[i]> := J[i].compute(rb[i-1],lb[i+1]);
    end-forall
  end-while
end
Mol. Dynamics with Spatial Decomposition

```
for i, j, k in cells
    <atoms[i,j,k]> := cells[i,j,k].produceAtoms();
end-for
for iter := 0 to MAX_ITER
    foreach i1,j1,k1,i2,j2,k2 in cellpairs
        <+forces[i1,j1,k1]> :=
            cellpairs[i1,j1,k1,i2,j2,k2].computeCoulombForces(  
                atoms[i1,j1,k1],atoms[i2,j2,k2]);
    end-for
    foreach ... for bonded forces.. Uses atoms and add to forces
        foreach i,j,k in cells
            <atoms[i,j,k]> := cells[i,j,k].integrate(forces[i,j,k]);
        end-for
end-for
```
Communication Patterns (3)

- Multicast: single produced value $\rightarrow$ multiple consuming objects

```
foreach i in A
    (points[i]) <- A[i].f(...);
end-foreach

foreach i,j in B
    (...) <- B[i,j].g(points[i]);
end-foreach
```

- Sequential code

```
AClass::f(..., outport points){
    Point local_points;
    local_points = ...;
    produce(points, local_points);
}
```
Communication Patterns (4)

- Scatter: a collection of produced values → chunked up
  → multiple consuming objects

```cpp
foreach i in A
    (points[i,*]) <- A[i].f(...);
end-foreach
foreach i,j in B
    (...) <- B[i,j].g(points[i,j]);
end-foreach
```

- Sequential code: local value at producing side has an additional dimension

```cpp
AClass::f(..., outport points){
    Point local_points[N];
    local_points = ...;
    produce(points, local_points);
}
```
Communication Patterns (5)

- Gather: multiple producing object → concatenated
  → single consuming objects

```plaintext
def gather(A, B):
    for i, j in A:
        points[i, j] = A[i, j].f(...);
    for j in B:
        (...)) = B[j].g(points[*, j]);
```

- Sequential code: consumed parameter has an additional dimension

```plaintext
BClass::g(Point *point[N]){  
    ...
}
```
Communication Patterns (6)

- Permutation operation: scatter + gather

```plaintext
foreach i in A
    (points[i,*]) <- A[i].f(...);
end-foreach
foreach j in B
    (...) <- B[j].g(points[*,j]);
end-foreach
```

- All-to-all operation, data transpose operation
  - 3D FFT
Parallel 3D FFT

foreach x in planes1
  (pencils[x,*]) <- planes1[x].fft1d();
end-foreach
foreach y in planes2
  planes2[y].fft2d(pencils[*,y]);
end-foreach

a) 1D FFT on Y Direction
b) Transpose with Pencils
c) 2D FFT in XZ Plane
Charisma Evaluation

- Performance
  - ARTS benefits
- Productivity
  - SLOC
  - Development time
- Application development experiences
  - LeanCP
  - Topology optimization
Performance and SLOC

PSC Cray XT3 System

NCSA Tungsten Cluster

2D Jacobi
(Size: 16384^2 on 4096 objects)

3D FFT
(Size: 512^3 on 256 objects)
## Performance and SLOC (2)

<table>
<thead>
<tr>
<th></th>
<th>Charisma</th>
<th>Charm++</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>253</td>
<td>354</td>
<td>28%</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>273</td>
<td>383</td>
<td>29%</td>
</tr>
<tr>
<td>Visualization</td>
<td>307</td>
<td>407</td>
<td>24%</td>
</tr>
<tr>
<td>Both</td>
<td>327</td>
<td>436</td>
<td>25%</td>
</tr>
</tbody>
</table>

**SLOC Comparison of Wator Code**

Screen Capture of Realtime Visualization of Wator
Development Time Reduction

Jacobi Development Time (Hours)

Wator Development Time (Hours)