Tammy Dahlgren, Tom Epperly, Scott Kohn, & Gary Kumfert
Goals

Describe our vision to the CCA

Solicit contributions (code) for:

- RMI (SOAP | SOAP w/ Mime types)
- Parallel Network Algs (general arrays)

Encourage Collaboration
Outline

Background on Components @llnl.gov
General MxN Solution : bottom-up
  Initial Assumptions
  MUX Component
  MxNRedistributable interface
Parallel Handles to a Parallel Distributed Component
Tentative Research Strategy
Components @llnl.gov

- Quorum - web voting
- Alexandria - component repository
- Babel - language interoperability
  maturing to platform interoperability
  - ... implies some RMI mechanism
  - SOAP | SOAP w/ MIME types
  - open to suggestions,
    & contributed sourcecode
Babel & MxN problem

Unique Opportunities
SIDL communication directives
Babel generates code anyway
Users already link against Babel Runtime Library
Can hook directly into Intermediate Object Representation (IOR)
Impls and Stubs and Skels

Application: uses components in user’s language of choice

Client Side Stubs: translate from application language to C

Internal Object Representation: Always in C

Server Side Skeletons: translates IOR (in C) to component implementation language

Implementation: component developers choice of language. (Can be wrappers to legacy code)
Out of Process Components

- Application
  - Stubs
  - IORs
  - IPC
- IPC
  - IORs
  - Skels
  - Impls
Remote Components

- Application
- Stubs
- IORs
- Marshaler
- Line Protocol

- Line Protocol
- Unmarshaller
- IORs
- Skels
- Impls
Outline

Background on Components @llnl.gov

General MxN Solution : bottom-up

Initial Assumptions

MUX Component

MxNRedistributable interface

Parallel Handles to a Parallel Distributed Component

Tentative Research Strategy
Initial Assumptions

Working Point-to-Point RMI
Object Persistence
Example #1: 1-D Vectors

```c
double d = x.dot( y );
```

```c
int globalSize = 6;
int localSize = 2;
int[ ] local2global = { 0, 1 };
double[ ] localData = { 0.0, 1.1 };
Vector y;
```

```c
int globalSize = 6;
int localSize = 2;
int[ ] local2global = { 2, 3 };
double[ ] localData = { 2.2, 3.3 };
Vector y;
```

```c
int globalSize = 6;
int localSize = 2;
int[ ] local2global = { 4, 5 };
double[ ] localData = { 4.4, 5.5 };
Vector y;
```

```c
int globalSize = 6;
int localSize = 3;
int[ ] local2global = { 3, 4, 5 };
double[ ] localData = { .6, .5, .4 };
Vector x;
double result;
```

```c
int globalSize = 6;
int localSize = 3;
int[ ] local2global = { 0, 1, 2 };
double[ ] localData = { .9, .8, .7 };
Vector x;
```

```c
double d = x.dot( y );
```
Example #1: 1-D Vectors

double d = x.dot( y );

p0
int globalSize = 6;
int localSize = 2;
int[ ] local2global = { 0, 1 };
double[ ] localData = { 0.0, 1.1 };

p1
int globalSize = 6;
int localSize = 2;
int[ ] local2global = { 2, 3 };
double[ ] localData = { 2.2, 3.3 };

p2
int globalSize = 6;
int localSize = 2;
int[ ] local2global = { 4, 5 };
double[ ] localData = { 4.4, 5.5 };

p3
int globalSize = 6;
int localSize = 3;
int[ ] local2global = { 0, 1, 2 };
double[ ] localData = { .9, .8, .7 };

p4
int globalSize = 6;
int localSize = 3;
int[ ] local2global = { 3, 4, 5 };
double[ ] localData = { .6, .5, .4 };
double vector::dot( vector& y ) {

// initialize
double * yData = new double[localSize];
y.requestData( localSize, local2global, yData);

// sum all x[i] * y[i]
double localSum = 0.0;
for( int i=0; i<localSize; ++i ) {
    localSum += localData[i] * yData[i];
}

// cleanup
delete[] yData;
return localMPIComm.globalSum( localSum );
}
Design Concerns

- Vector y is not guaranteed to have data mapped appropriately for dot product.
- Vector y is expected to handle MxN data redistribution internally.

```c
y.requestData( localSize, local2global, yData);
```

Should each component implement MxN redistribution?
Outline

- Background on Components @llnl.gov
- General MxN Solution: bottom-up
  - Initial Assumptions
  - MUX Component
    - MxNRedistributable interface
  - Parallel Handles to a Parallel Distributed Component
- Tentative Research Strategy
double vector::dot( vector& y ) {

    // initialize
    MUX mux( *this, y );
    double * yData =
        mux.requestData( localSize, local2global );

    // sum all x[i] * y[i]
    double localSum = 0.0;
    for( int i=0; i<localSize; ++i ) {
        localSum += localData[i] * yData[i];
    }

    // cleanup
    mux.releaseData( yData );
    return localMPIComm.globalSum( localSum );
}
vector<T>::parallelOp( vector<T>& y ) {

    // initialize
    MUX mux( *this, y );
    vector<T> newY =
        mux.requestData( localSize, local2global );

    // problem reduced to a local operation
    result = x.localOp( newY );

    // cleanup
    mux.releaseData( newY );
    return localMPIComm.reduce( localResult );
}
Rule #2: MUX distributes data

Users invoke parallel operations without concern to data distribution

Developers implement local operation assuming data is already distributed

Babel generates code that reduces a parallel operation to a local operation

MUX handles all communication

How general is a MUX?
Example #2: Undirected Graph
Key Observations

Every Parallel Component is a container and is divisible to subsets.
There is a minimal (atomic) addressable unit in each Parallel Component.
This minimal unit is addressable in global indices.
Atomicity

Vector (Example #1):
atom - scalar
addressable - integer offset

Undirected Graph (Example #2):
atom - vertex with ghostnodes
addressable - integer vertex id

Undirected Graph (alternate):
atom - edge
addressable - ordered pair of integers
Outline

Background on Components @llnl.gov
General MxN Solution : bottom-up
  Initial Assumptions
  MUX Component
  MxNRedistributable interface
Parallel Handles to a Parallel Distributed Component
  Tentative Research Strategy
MxNRedistributable Interface

```java
interface Serializable {
    store( in Stream s );
    load( in Stream s );
};

interface MxNRedistributable extends Serializable {
    int getGlobalSize();
    local int getLocalSize();
    local array<int,1> getLocal2Global();

    split ( in array<int,1> maskVector, 
        out array<MxNRedistributable,1> pieces);
    merge( in array<MxNRedistributable,1> pieces);
};
```
Rule #3: All Parallel Components implement “MxNRedistributable”

Provides standard interface for MUX to manipulate component

Minimal coding requirements to developer

Key to abstraction

  split()

  merge()

Manipulates “atoms” by global address
Now for the hard part...

... 13 slides illustrating how it all fits together for an Undirected Graph
`%> mpirun -np 2 graphtest`

pid=0

pid=1
BabelOrb * orb =
BabelOrb.connect( "http://...");
Graph * graph = orb->create("graph",3);
graph->load("file://...");
graph->doExpensiveWork();

CASC
PostProcessor * pp = new PostProcessor;
pp->render( graph );

MUX queries graph for global size (12)

Graph determines particular data layout (blocked)

MUX is invoked to guarantee that layout before render implementation is called
MUX solves general parallel network flow problem (client & server)
MUX opens communication pipes

orb pp
graph require
MUX

0 1 2 3 4 5

0, 1, 2, 3
4, 5
6, 7
8, 9, 10, 11
MUX splits graphs with multiple destinations (server-side)
MUX sends pieces through communication pipes (persistance)
MUX receives graphs through pipes & assembles them (client side)
pp -> render_impl( graph );
(user’s implementation runs)
Outline

Background on Components @llnl.gov
General MxN Solution : bottom-up
Initial Assumptions
MUX Component
MxNRedistributable interface

Parallel Handles to a Parallel Distributed Component
Tentative Research Strategy
Summary

All distributed components are containers and subdivisable

The smallest globally addressable unit is an atom

MxNRedistributable interface reduces general component MxN problem to a 1-D array of ints

MxN problem is a special case of the general problem N handles to M instances

Babel is uniquely positioned to contribute a solution to this problem
Outline

- Background on Components @llnl.gov
- General MxN Solution: bottom-up
  - Initial Assumptions
  - MUX Component
  - MxNRedistributable interface
- Parallel Handles to a Parallel Distributed Component

Tentative Research Strategy
Tentative Research Strategy

Fast Track
- Java only, no Babel
- serialization & RMI built-in
- Build MUX
- Experiment
- Write Paper

Sure Track
- Finish 0.5.x line
- add serialization
- add RMI
- Add in technology from Fast Track
Open Questions

- Non-general, Optimized Solutions
- Client-side Caching issues
- Fault Tolerance
- Subcomponent Migration
- Inter vs. Intra component communication
- MxN, MxP, or MxPxQxN
MxPxQxN Problem

Long-Haul Network
MxPxQxN Problem

Long-Haul Network
The End
Work performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48