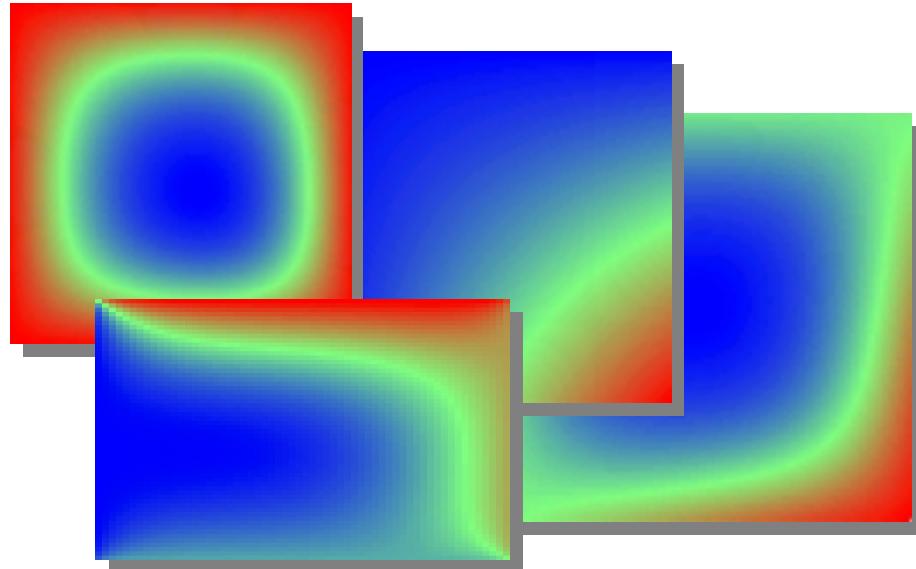


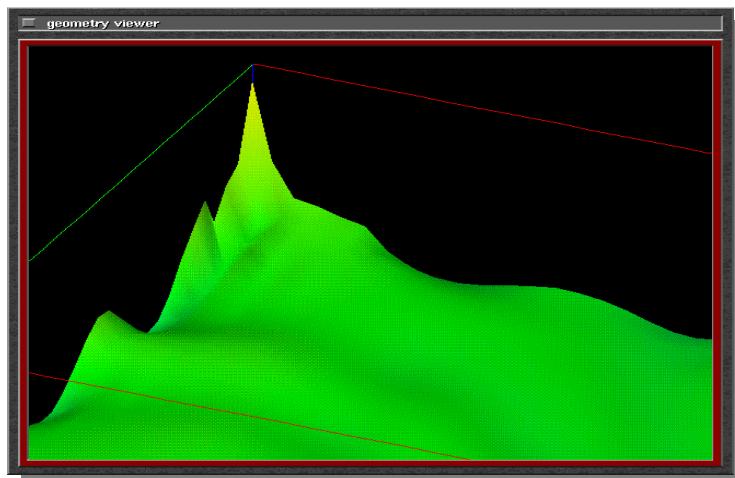
# CUMULVS Tutorial

ACTS Collection Workshop



Dr. James Arthur Kohl

Computer Science and Mathematics Division  
Oak Ridge National Laboratory



August 26, 2005



# Scientific Simulation Issues...

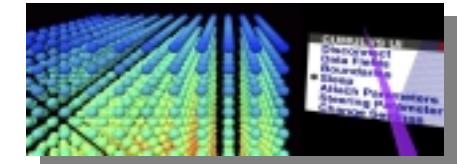
- Fundamental Parallel Programming
  - Synchronization, Coordination & *Control*
- Distributed Data Organization
  - Locality, Latency Hiding, *Data Movement*
- Long-Running Simulation Experiments
  - *Monitoring, Fault Recovery*
- Massive Amounts of Data/Information
  - Archival Storage, *Visualization*
- Too Much Computer, Not Enough Science!
  - *Need Some Help...*

# Potential Benefits from Computer Science Infrastructure:

- **On-The-Fly Visualization**

⇒ Intuitive Representation of Complex Data

⇒ Interactive Access to Intermediate Results



- **Computational Steering**

⇒ Apply Visual Feedback to Alter Course/Restart

⇒ “Close Loop” on Experimentation Cycle

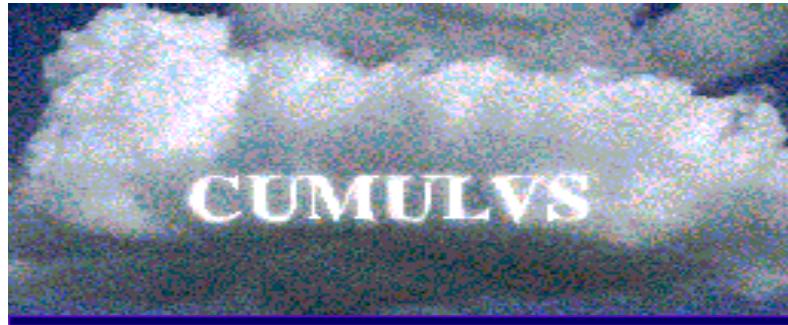


- **Fault Tolerance**

⇒ Automatic Fault Recovery, Load Balancing

⇒ “Keep Long-Running Simulations Running Long” ☺





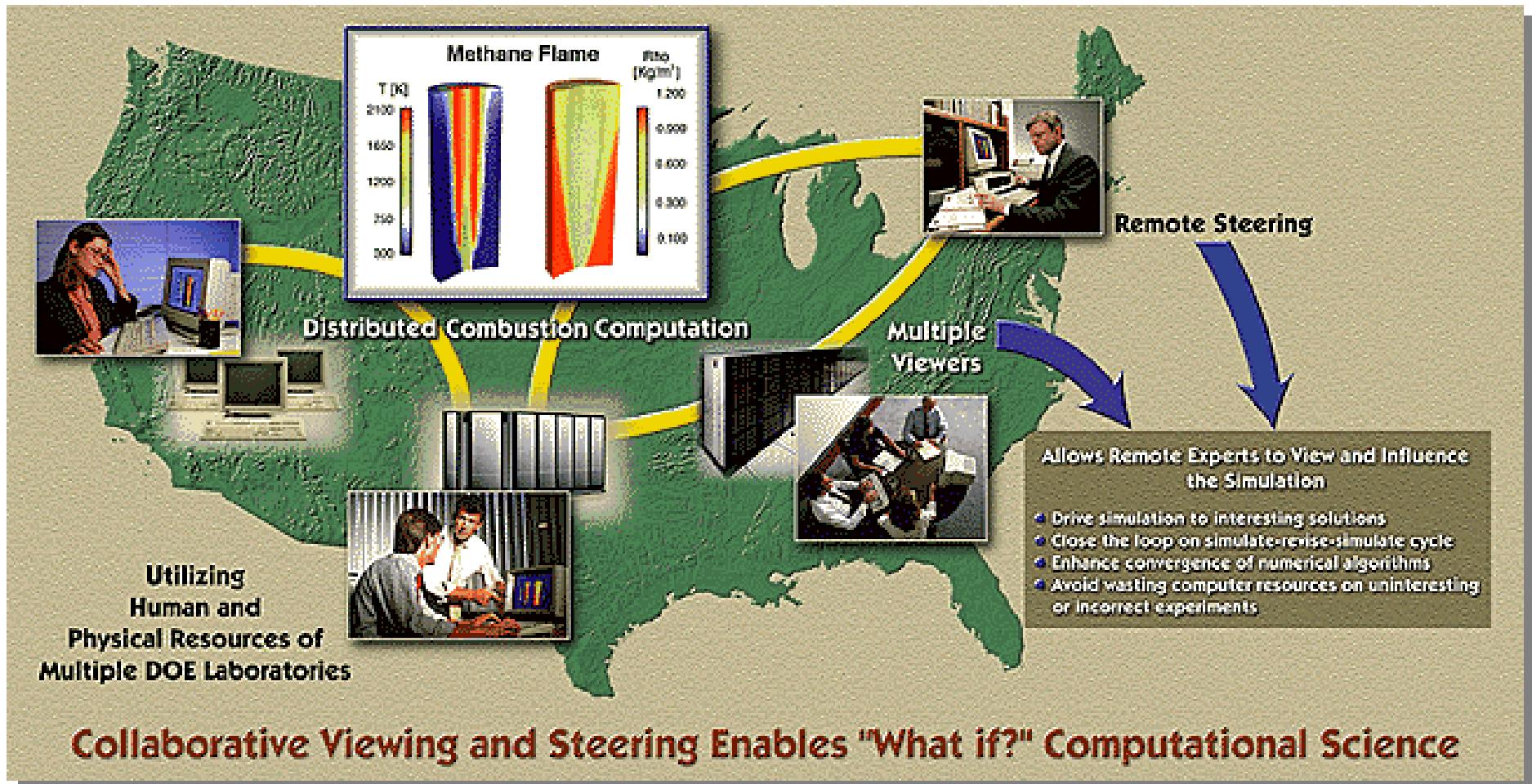
(Not a *Climate* Modeling or Simulation Tool...! ☺)

“Collaborative, User Migration, User Library for Visualization and Steering”

- Collaborative Infrastructure for Interacting with Scientific Simulations:
  - ⇒ Run-Time Visualization by Multiple Viewers
    - Dynamic Attachment as Needed, Independent Views
  - ⇒ Coordinated Computational Steering
    - Model & Algorithm Parameters
  - ⇒ Heterogeneous Checkpointing/Fault Tolerance
    - Monitoring & Data Recovery from Faults, Task Migration
  - ⇒ Coupled Parallel Models, Data Redistribution

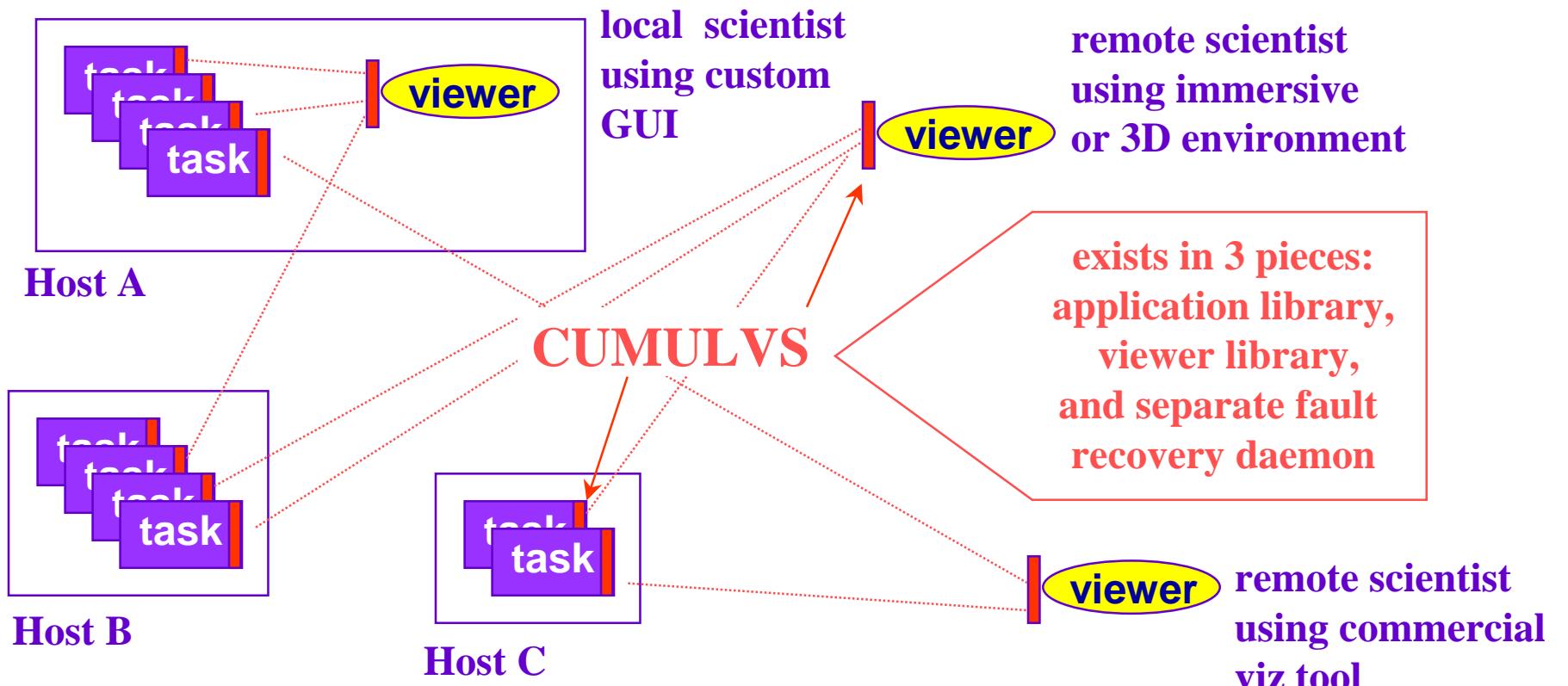


# Collaborative Combustion Simulation



# CUMULVS Architecture

coordinate the consistent collection and dissemination of information to/from parallel tasks to multiple viewers

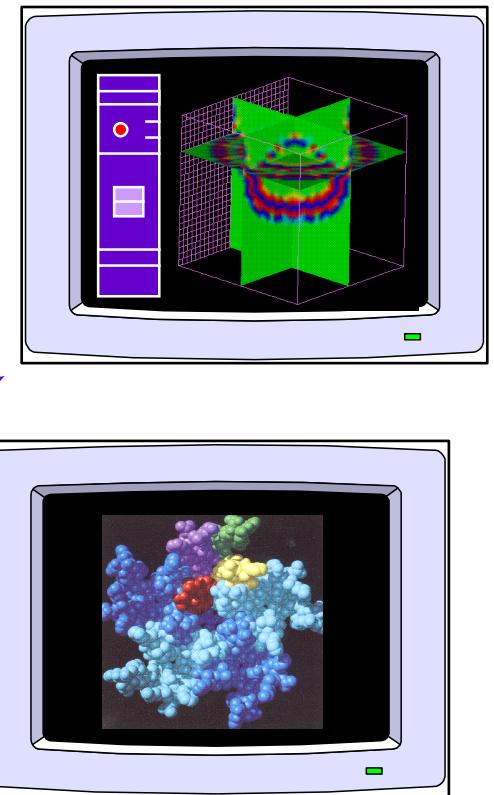


interact with live distributed/parallel software simulation  
supports most target platforms (PVM, MPI, ...)

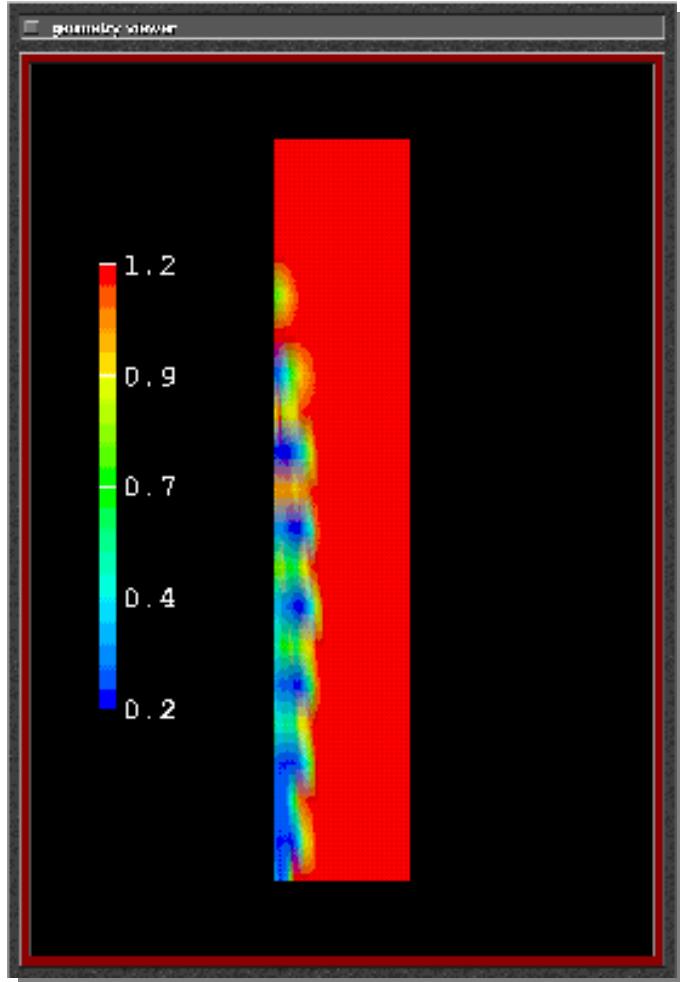
Kohl/2005-6

# CUMULVS Visualization Features

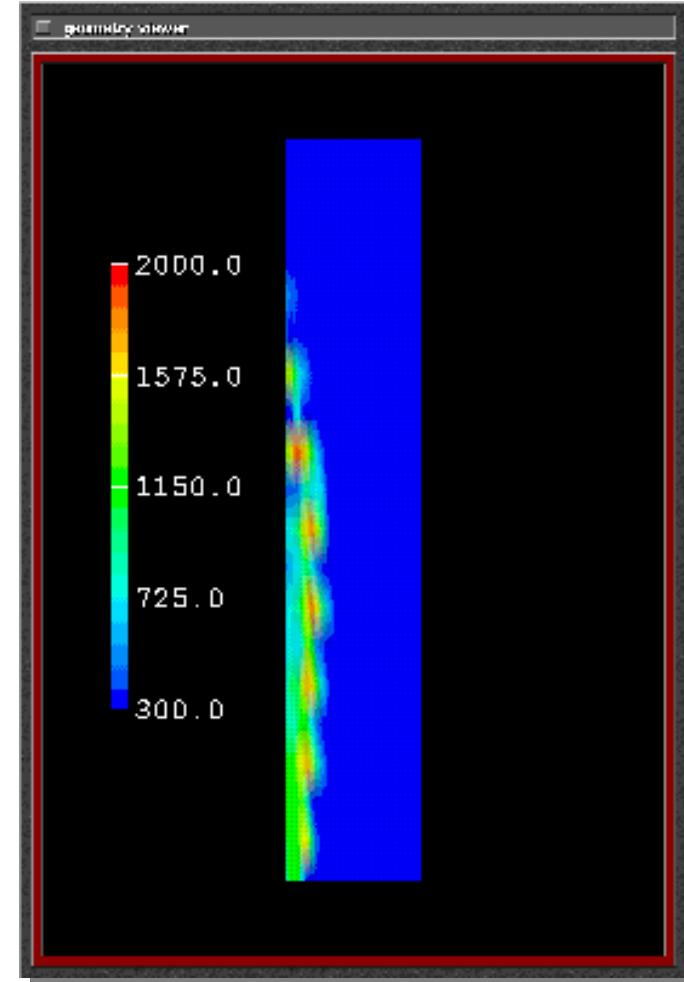
- **Interactive Visualization**
  - ⇒ Simple API for Scientific Visualization
  - ⇒ Apply To Your Favorite Visualization Tool
- **Minimize Overhead When No Viewers**
  - ⇒ One Message Probe, No Application Penalty
- **Send Only Viewed Data**
  - ⇒ Partial Sub-Array/Resolution Control
- **Rect Mesh & Particle Data**
- **Common (HPF) Data Distributions**
  - ⇒ BLOCK, CYCLIC, EXPLICIT, COLLAPSE



# Multiple Simultaneous Views

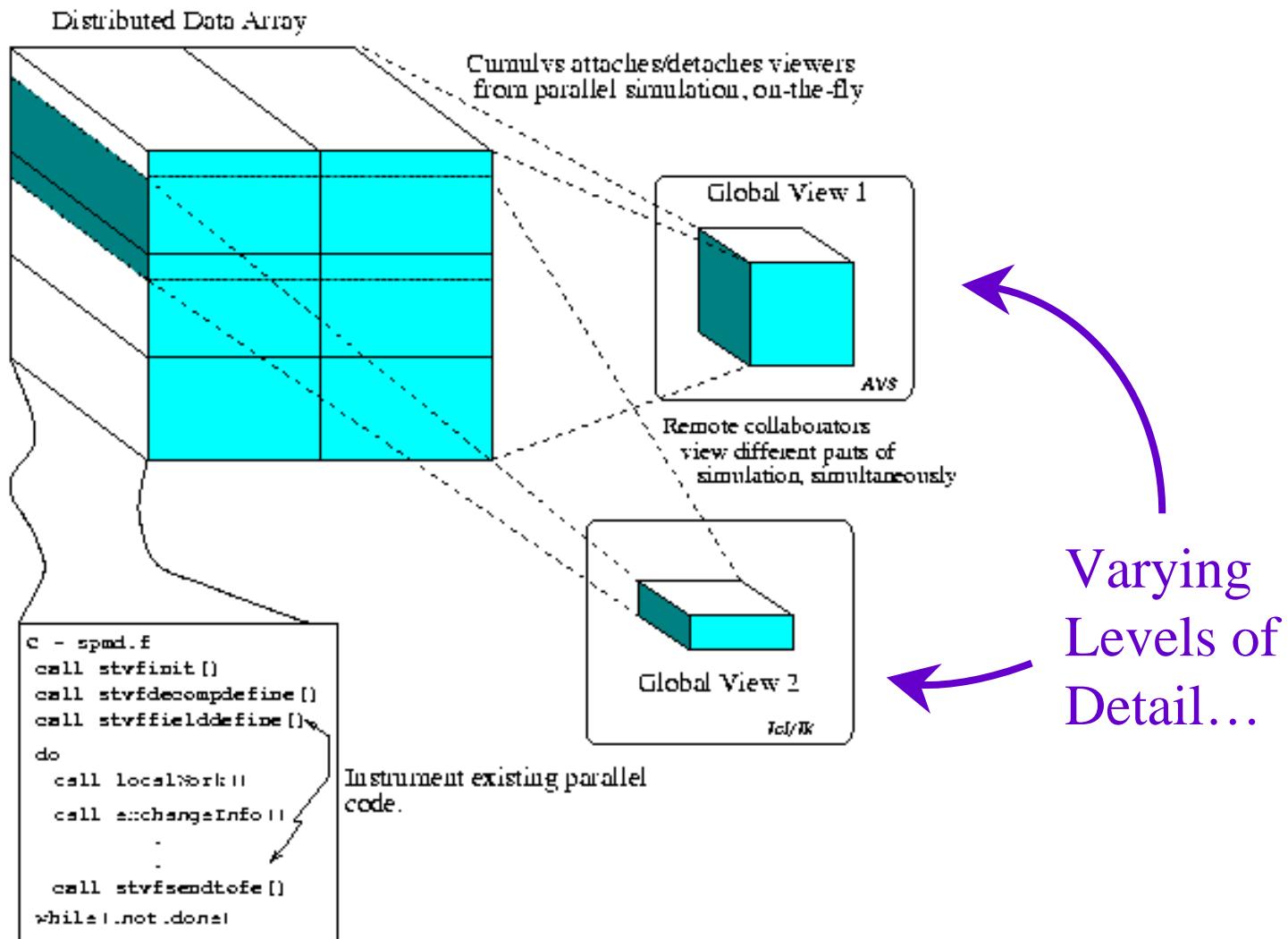


Density



Temperature

# Multiple Distinct Views



# Examples of CUMULVS Calls

## Application Side Library

```
#include "stv.h"

stvInit( &argc, &argv,
         "csimple", STVTAG, NTASKS, myInstance );

decompId = stv_decompDefine( 3,
                             axisType[], axisInfo1, axisInfo2,
                             glb[], gub[], prank, pshape[] );

fieldId = stv_fieldDefine( density, "density",
                           decompId, fieldOffsets[], declared[], stvFloat,
                           procAddr[], stvVisOnly );

. . . stv_sendReadyData( stvSendDefault );
```

# Examples of CUMULVS Calls

## Viewer Side Library

```
#include "stv.h"
#include "stv_glob.h"

STV_VIEWER VIEWER, STV_VIEW_FIELD VF, STV_VIEW_FIELD_GROUP VFG;

stv_viewer_init( &VIEWER, appname, blocking );

VF = stv_get_view_field_name( fieldName,
    VIEWER->vfields, VIEWER->nvfields );
VF->selected = stvTrue;
VF->view_type = stvFloat;
VF->view_storage_order = stvRowMajor;
. . .

VFG = stv_viewer_submit_field_request( VIEWER, visRegions[], freq,
    stvFieldRqstDefault, &restart );

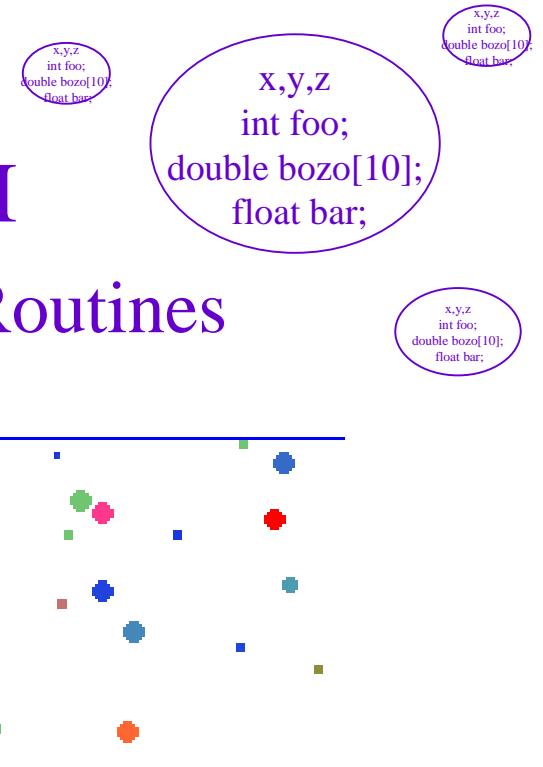
do {
    stv_viewer_receive_frame( &VFG, VIEWER, &restart, stvMsgNoBlock );

    ptr = VFG->field_values[ fieldIndex ][ regionIndex ];
    fval = STV_VALUE_OF( ptr, float );
} while ( . . . );

stv_viewer_release_field( &VFG, VIEWER );
```

# CUMULVS Particle Handling

- Particle Data Fundamentally Different
  - ⇒ Data Fields Encapsulated in a Particle Container
  - ⇒ Explicit Coordinates Per Particle
- Particle-Based Decomposition API
  - ⇒ User-Defined, Vectored Accessor Routines
- Viewing Particle Data
  - ⇒ AVS Module Extensions
  - ⇒ Tcl/Tk Slicer Particle Mode



# Example Particle API

```
chargeId = stv_particleDefine( "charge",
  2, glb[], gub[], NTASKS,
  get_charge(), (STV_VALUE) NULL,
  (STV_MAKE_PARTICLE) NULL, (STV_VALUE) NULL );

stv_pfieldDefine( "velMagn", chargeId, stvFloat, nElems,
  get_charge_pfield(), (STV_VALUE) PF_VEL_MAGN,
  (STV_SET_PFIELD) NULL, (STV_VALUE) NULL,
  stvVisOnly );

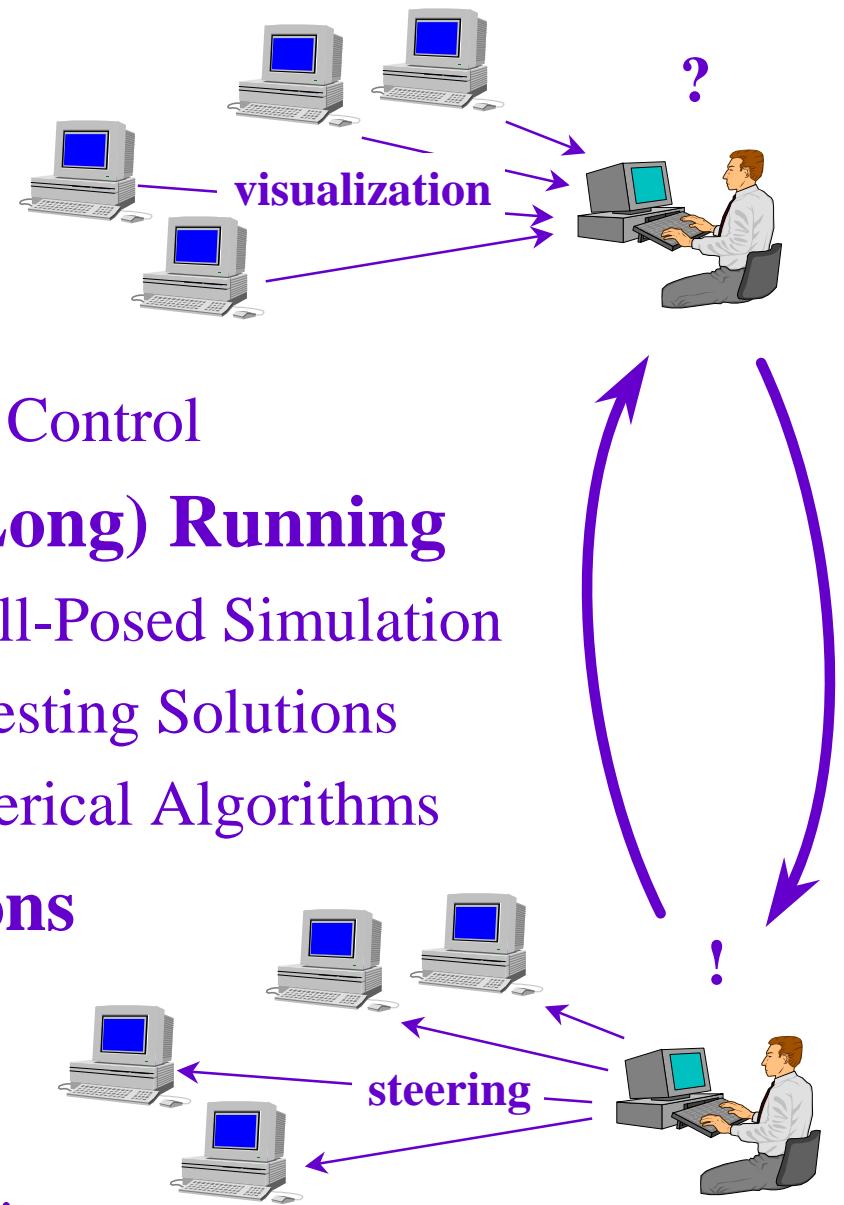
void get_charge( int chargeIndex, STV_REGION Region,
  STV_VALUE clientData, STV_PARTICLE_ID *id,
  int *coords );

void get_charge_pfield( STV_PARTICLE_ID id,
  STV_VALUE fieldId, STV_VALUE *value, int *nelems );
```

# CUMULVS

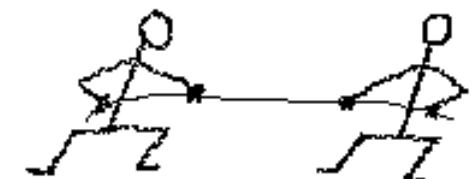
## Steering Features

- **Computational Steering**
  - ⇒ API for Interactive Application Control
- **Modify Parameters While (Long) Running**
  - ⇒ Eliminate Wasteful Cycles of Ill-Posed Simulation
  - ⇒ Drive Simulation to More Interesting Solutions
  - ⇒ Enhance Convergence of Numerical Algorithms
- **Allows “What If” Explorations**
  - ⇒ Closes Loop of Standard  
Simulation Cycle
  - ⇒ Explore Non-Physical Effects...



# Coordinated Steering

- Multiple, Remote Collaborators (“Steerers”)
- Simultaneously Steer Different Parameters
  - ⇒ Physical Parameters of Simulation
  - ⇒ Algorithmic Parameters ~ e.g. Convergence Rate
- Cooperation Among Collaborators...
  - ⇒ Parameter Locking Prevent Conflicts
  - ⇒ Vectored/Indexe Parameters...
- Parallel/Distributed Simulations
  - ⇒ Synchronize with Parallel Tasks
  - ⇒ All Tasks Update Each Parameter in Unison



# Example Steering Parameter Definitions

```
paramId = stv_paramDefine( "probability",
    (STV_VALUE) &prob, stvDouble, stvVisOnly );

paramId = stv_vparamDefine( "eField",
    STV_VALUE field_params[], char *field_names[],
    field_types[], 3, stvVisOnly );

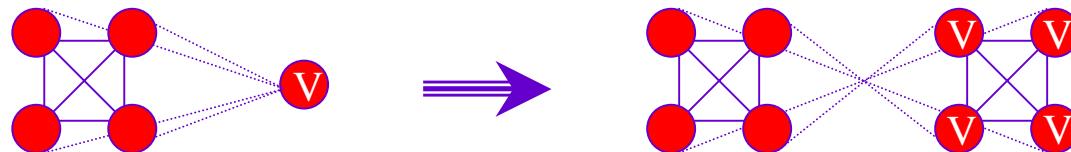
...nSteered = stv_sendReadyData( stvSendDefault );

if( stv_isParamChanged( paramId ) )
    ...
```

# Parallel Model Coupling in CUMULVS

- Natural Extension to CUMULVS Viewer Scenario

⇒ Promote “Many-to-1” → “Many-to-Many”



Appl Tasks  
as “Viewers”  
onto Coupled  
Appl Tasks...

- Translate Disparate Data Decompositions

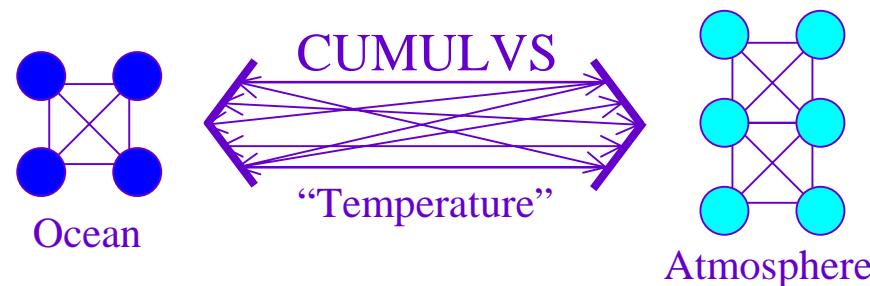
⇒ Parallel Data Redistribution Among Common Data Fields

→ `stv_couple_fields( fieldID, appname, fieldname, ... );`

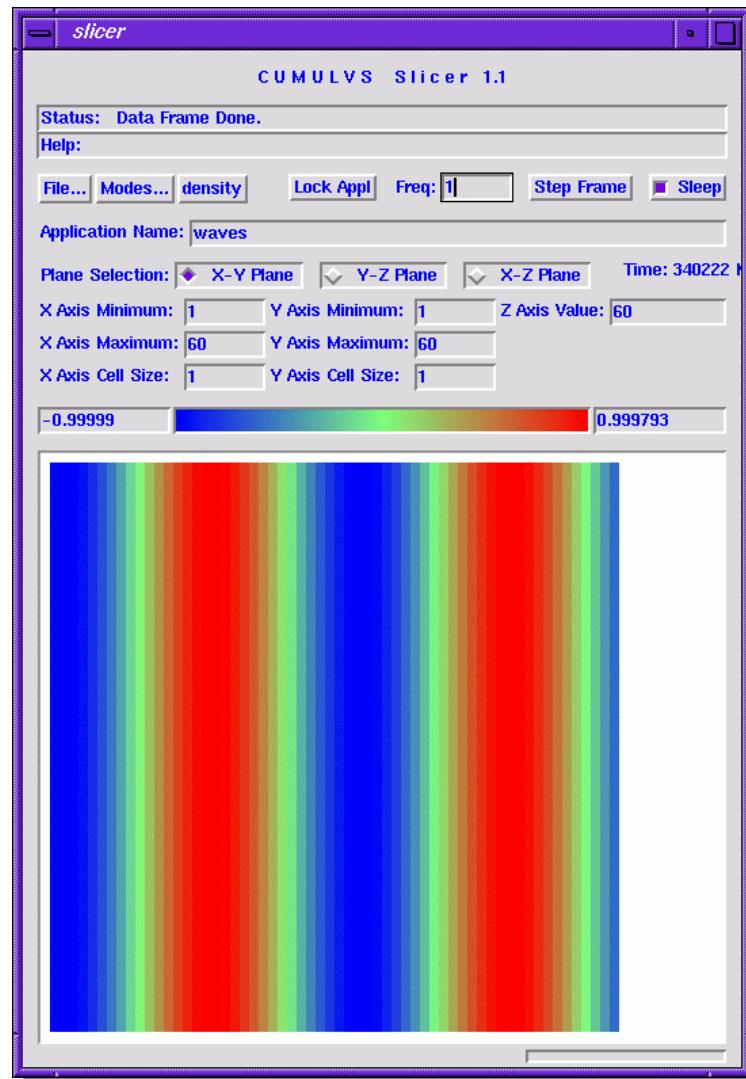
⇒ Fundamental Model Coupling Capability (Data Exchange)

→ Next Steps ~ Interpolation in Space & Time, Flux Conserve, Units...

E.g. Regional Climate Modeling:

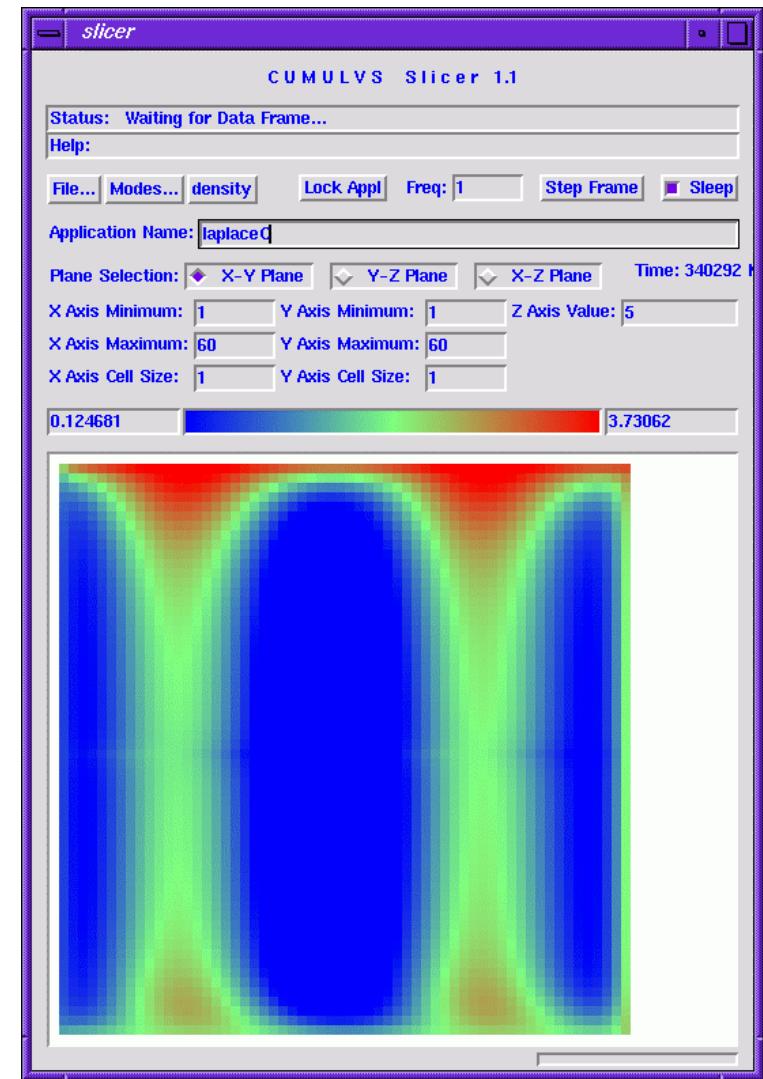


# Simple CUMULVS Coupling Example



ORNL

Sinusoidal Forcing Function



Live Surface Boundary

Kohl/2005-18

# Simple Coupling Code

```
coupleId = stv_couple_fields( fieldId,
    "wavesAppl", "density",
    DST_CFREQ, SRC_CFREQ,
    dstNoSynch, srcNoSynch, blocking );

...stv_sendReadyData( stvSendDefault );

stv_couple_finished( coupleId );
```

# Instrumenting Programs for CUMULVS

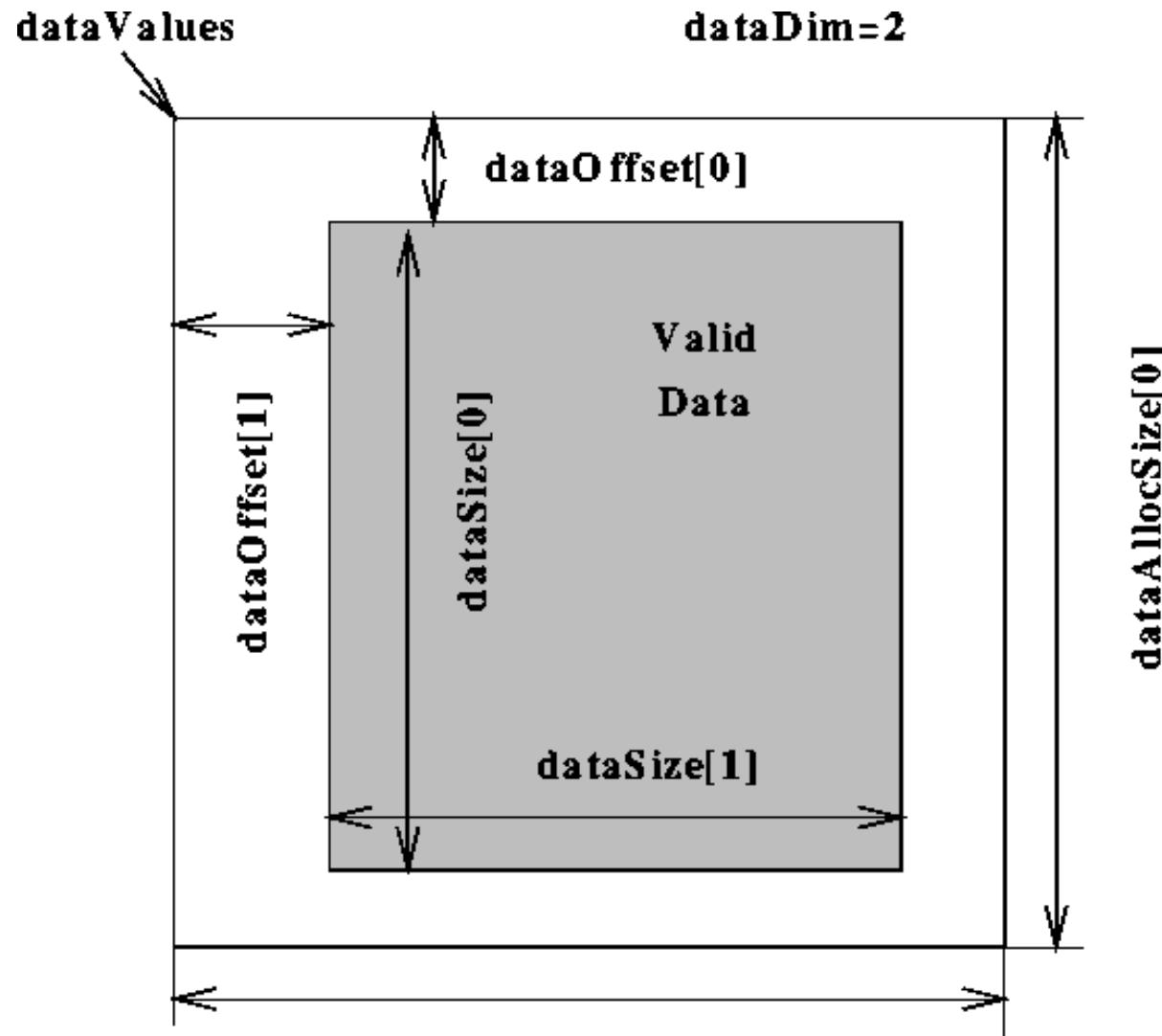
- CUMULVS Initialization ~ One Call (Per Task)
  - ⇒ Logical Application Name, Total # of Tasks
- Data Fields (for Visualization & Checkpointing)
  - ⇒ Local Allocation: Name, Type, Size, Offsets
  - ⇒ Data Distribution: Dim, Decomp, Processor Topology
- Steering Parameters
  - ⇒ Logical Name, Data Type and Pointer
- Periodic CUMULVS Handler
  - ⇒ Pass Control for Transparent Access/Processing
- Typically *10s* of Lines of Code at Most... ☺



# Decompositions and Data Fields

- Decomposition = Data Distribution “Template”
  - ⇒ Same Decomp can be Re-used for Many “Fields”
  - ⇒ Describes How “Global” Array is Broken Up
    - Overall Global Coordinates, Block vs. Cyclic Patches,  
Mapping of Patches to Processors...
- Data Fields Associate Actual Data with Decomp
  - ⇒ “Name”, Data Pointer and Type, Local Allocation,  
Position in Processor Decomposition...

# Local Allocation Organization



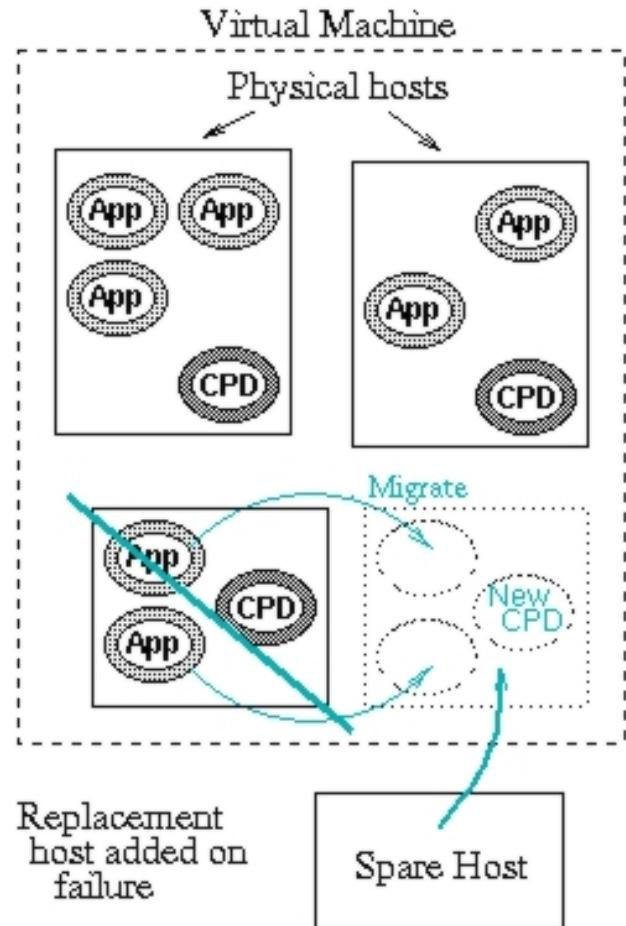
# CUMULVS Fault Tolerance Features

- **Application Fault Tolerance**
  - ⇒ Automatic Detection and Recovery from Failures
- **User Directed Checkpointing**
  - ⇒ User Decides What/Where to Checkpoint
  - ⇒ Minimizes Stored Data, Adds Semantics
- **Heterogeneous Task Migration**
  - ⇒ Restart Tasks on Heterogeneous Hosts
  - ⇒ Restart can be Automatically *Repartitioned* if New Hosts of Different Size/Topology (Burnt Offering Required ☺)
- **Avoids Synchronizing Distributed Tasks**
  - ⇒ Asynchronous Checkpoint Collection and Fault Detection
  - ⇒ Minimize Intrusion of Checkpoint/Restart



# Run-Time Fault Monitor

- One Checkpointing Daemon (CPD) Per Host
  - ⇒ Ckpt Collector/Provider
  - ⇒ Run-Time Monitor
  - ⇒ Console for Restart/Migrate
- CPDs Comprise Fault-Tolerant Application...
  - ⇒ Handle Failure of Host/CPD
  - ⇒ Coordinate Redundancy
  - ⇒ Ring Topology



# Rollback Versus Restart...

- Rollback Recovery:
  - ⇒ Only Replace Failed Tasks, “Roll Back” the Rest
  - ⇒ Elegant & “Cool”, But You Must...
    - Monitor ALL Communication for Restart Notification
    - Unroll Program Stack, Reset Comm & File Pointers...
  - ⇒ *Necessary* for High Overhead Restart Cases
- Restart Recovery:
  - ⇒ “Genocide” ~ Kill Everything & Restart All Tasks
  - ⇒ Simple Approach, No Additional Instrumentation
  - ⇒ Not as Efficient a Recovery in All Cases...

# Checkpoint Data Collection

- Data from Each Local Task Collected/Committed
  - `stv_checkpoint();`
- Invoke When Parallel Data / State “Consistent”...
  - ⇒ Highly Non-Trivial in General! (Chandy/Lamport)
  - ⇒ Straightforward for Most Iterative Applications
    - Save Checkpoint at Beginning or End of Main Loop
- No Automatic Capturing of Other Internal State:
  - ⇒ Open Files, I/O, Messages-in-Transit...
  - ⇒ CUMULVS Assumes User Handles This Recovery
    - Can Be Done Manually Using Saved Checkpoint State
    - Future Extensions to Assist...

# Manual Software Instrumentation

- SPDT 98 Case Study ~ SW Instrumentation Cost

Instrumentation:	Seismic:	Wing Flow:
Original Lines of Code	20,632	2,250
Vis / Steer System Init	3	3
Vis / Steer Variable Decl	48	73
CP Restart Initialization	21	12
CP Rollback Handling	41	34
Total Instrumentation	204 ~ 1.0 %	188 ~ 7.7 %

# Checkpointing Efficiency

- SPDT 98 Case Study ~ Execution Overhead

Seconds per Iteration

Experiment:	SGI:	Cluster:	Hetero:
Seismic - No Checkpointing	2.83	6.23	9.46
Seismic - Checkpoint for Restart	2.99	6.50	10.76
Seismic - Checkpoint for Rollback	3.03	6.66	10.90
Wing - No Checkpointing	0.69	1.58	6.14
Wing - Checkpoint for Restart	0.77	1.71	7.10
Wing - Checkpoint for Rollback	0.79	1.71	7.30

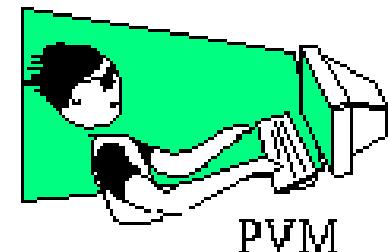
(Checkpointing Every 20 Iterations ~ every *15 sec to 4 mins...!*)

**Seismic Overhead: 4-14% Restart, +1-3% Rollback**

**Wing Overhead: 8-15% Restart, +0-2.5% Rollback**

# CUMULVS Run-Time System

- Originally built on PVM (Parallel Virtual Machine)
- MPI Standard has evolved, become proliferate...
  - ⇒ Many other available communication subsystems:
    - OpenMP, ECho (event-based), Linda (tuple-based), etc...
- CUMULVS “*works*” with these systems...
  - ⇒ If you carry PVM around, too...! ☺
- “Native” solutions would be “nice”...
  - ⇒ Hmmmm...



# Supporting MPI Applications in CUMULVS

- CUMULVS Worked “Fine” with MPI Applications... ☺
- But MPI Didn’t Have Everything We Needed (Internally)
  - ⇒ Static Model, Minimal Operating Environment
  - ⇒ No Name Service/Database; No Fault Recovery/Notification
  - ⇒ MPI\_Spawn( )...? Need Proxy Server for Viewer Attachment?
- Previous Solution:
  - ⇒ Applications Communicate Using MPI or PVM or ???
  - ⇒ CUMULVS Viewers and CPDs Still Attach Using PVM...
- *New Communication Substrate Under Development!*
  - ⇒ Provide Viz/Steering Features to MPI via Generic TCP/IP!

# New Multi-Communication Run-time Environment!

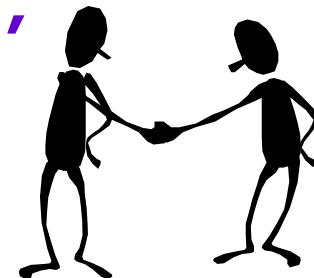


- Supports PVM, MPI *and* TCP Connections
- Simultaneous Use of ANY Subsystem!
  - ⇒ Interactive Selection Based on Availability
    - or Connectivity, such as with MPP systems...
  - ⇒ Application Publishes “Address” for ALL Systems
  - ⇒ Viewer Programs Query Substrates as Desired
    - Each Viewer can Connect Independently...
- Minimal CUMULVS API Changes...
  - ⇒ Mostly Viewers; *Lots* of Internal Changes... :-o

# Generic Application Discovery API

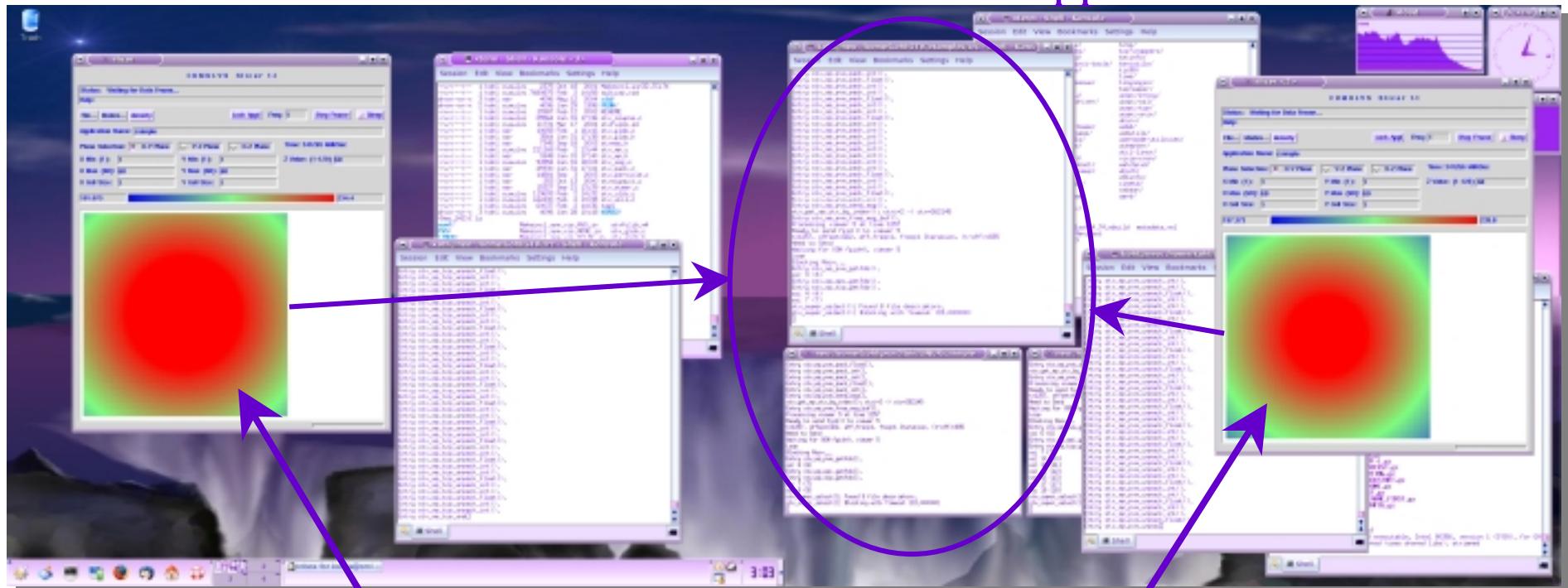
- Use Lowest Common Denominator ~ “Files”... ☺
- Rendezvous via SSH:
  - ⇒ Select “Globally Visible” Spot ~ User@Host:/Directory...
  - ⇒ Directory must be *Writable* by that User...
- Specify Via Environment Variables:
  - ⇒ **STV\_MP\_RSH**, **STV\_MP\_APPHOST**, **STV\_MP\_APPUSER** and **STV\_MP\_APPDIR**...
- Or, Specify Via Programmatic Interface:

```
stv_set_appl_discovery( char *apprsh,
                           char *apphost, char *appuser,
                           char *appdir );
```



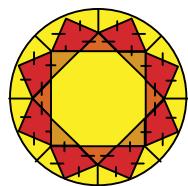
# Multi-Messaging Viewer Prototype!

Same Parallel Appl



Connected via TCP

Connected via PVM

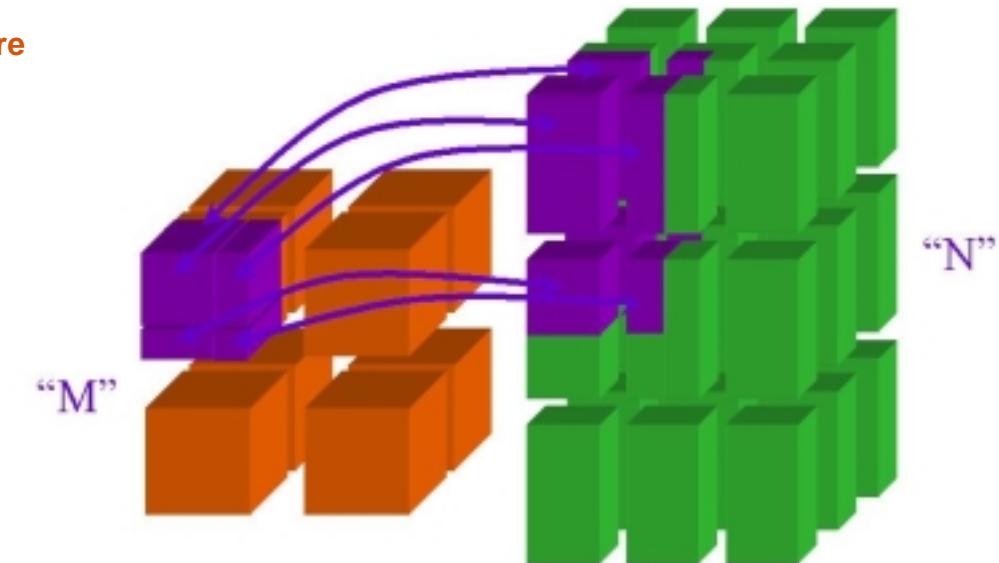


**CCA**

Common Component Architecture

Now Called  
“Parallel  
Coupling  
Infrastructure”  
(PCI)...

# CUMULVS Technology Integration



- **CCA “MxN” Parallel Data Redistribution**
  - ⇒ Builds on CUMULVS Viz & Coupling Protocols
  - ⇒ Also Based on PAWS (LANL)
    - “MxN” Generalizes Capabilities of Both Systems
      - \* Point-to-Point versus Persistent Connections (a la Viz)
    - CUMULVS Viz/Steering Complements PAWS Parallel Coupling Work

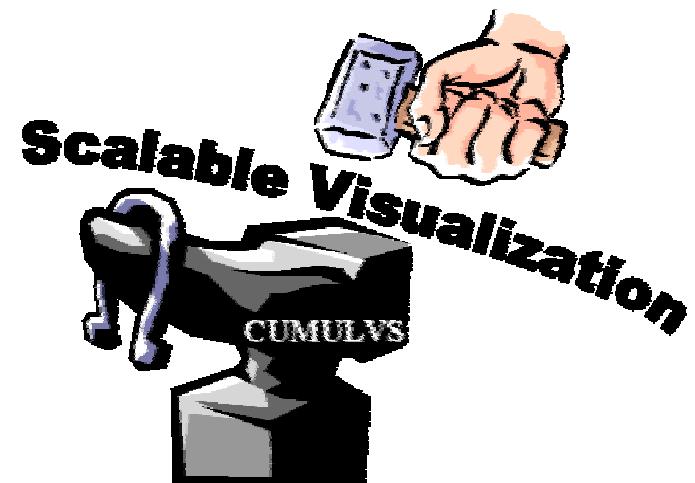
# CUMULVS as a Foundry to Forge New Technology

## High-Performance Visualization

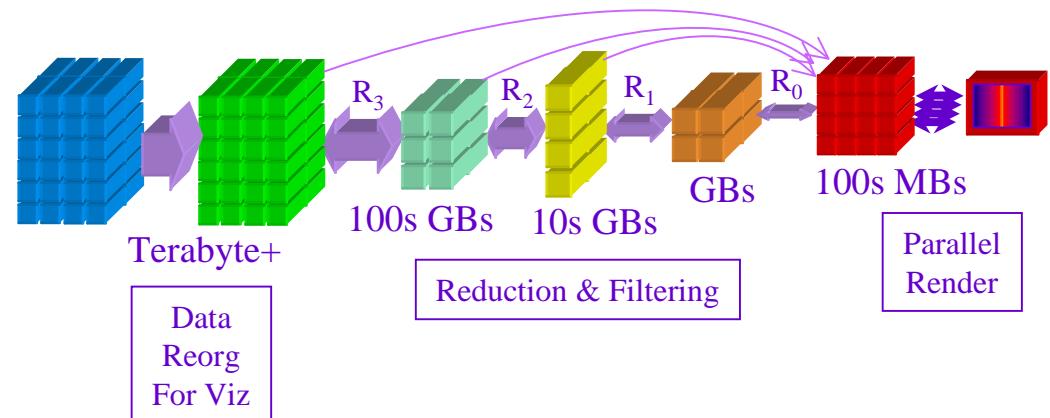
- Full Parallel Integration of Pipeline
- CCA “MxN” → “M x N x P x Q x R”!



Proposed “Fully Connected”  
User-Centric Simulation Cycle

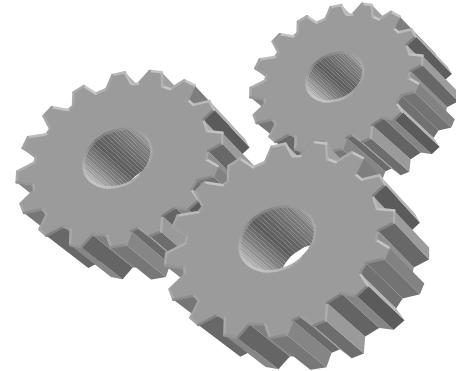


“Data”  
Scalable Visualization Cache Architecture



# Future CUMULVS

## Feature Extensions...



- Componentization of Functions:
  - ⇒ CCA Components for Viz, Steering and Fault Tolerance
    - For User Components and Framework Services...
- Clean Up Mapping to Other Messaging Substrates
  - ⇒ Functional, but not “Bullet-Proof”...
- Checkpointing Efficiency:
  - ⇒ Tasks Write Data in Parallel / Parallel File System?
  - ⇒ Redundancy Levels, Improve Scalability...

# CUMULVS Summary

- Interact with Scientific Simulations
  - ⇒ Dynamically Attach Multiple Visualization Front-Ends
  - ⇒ Steer Model & Algorithm Parameters On-The-Fly
  - ⇒ Automatic Heterogeneous Fault Recovery & Migration
  - ⇒ Couple Disparate Simulation Models
- Application Opportunities
  - ⇒ Forging Scalable Visualization Technologies
  - ⇒ “MxN”/PCI, Viz, Steering, F.T. ~ Components in CCA

<http://www.csm.ornl.gov/cs/cumulvs.html>

Seismic Example ~ 2D (Tcl/Tk)

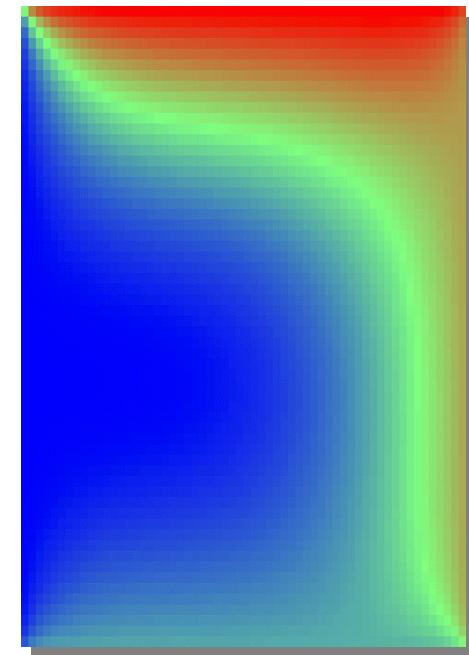
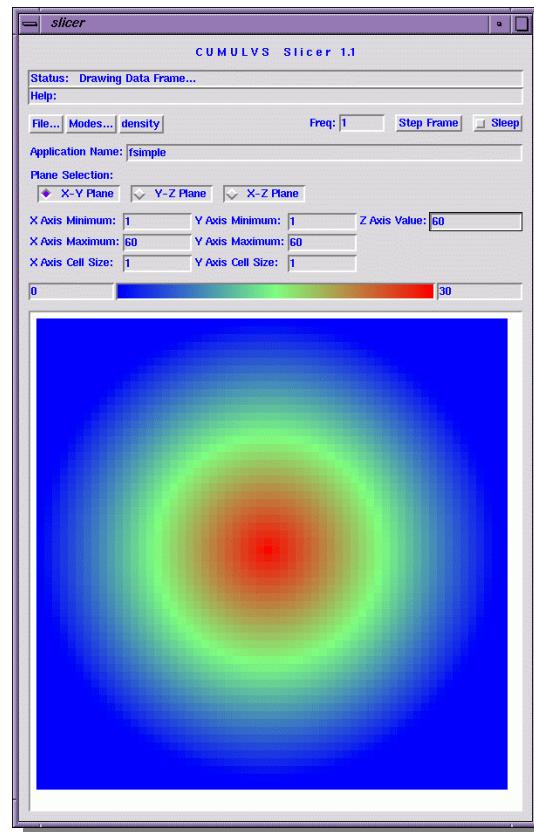
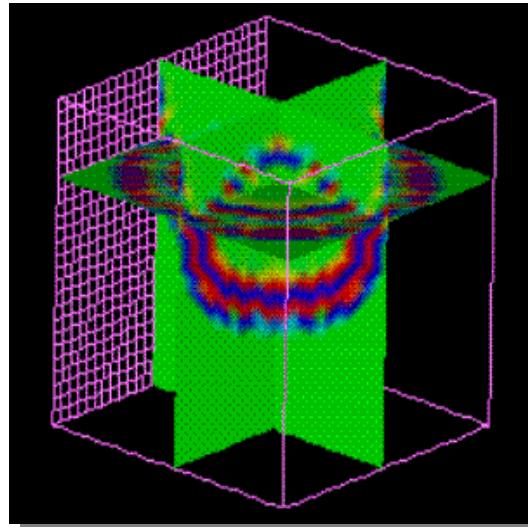
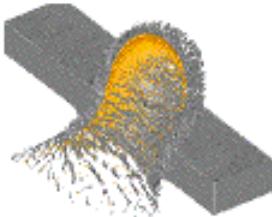
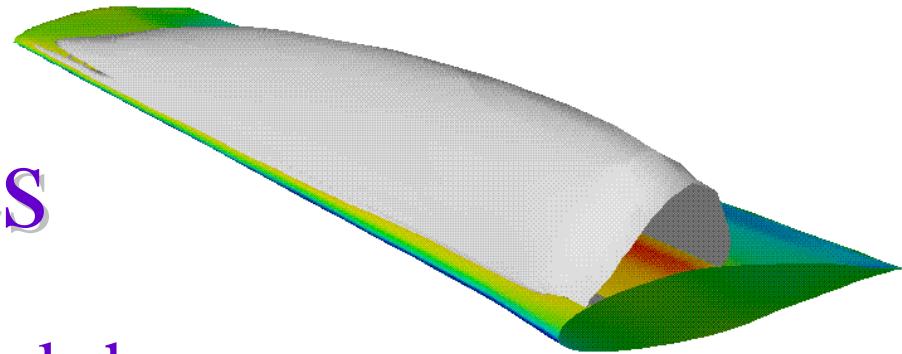
Seismic Example ~ 3D (AVS)

Air Flow Over Wing Example ~ 3D (AVS)

# CUMULVS

## User Guide Notes

ACTS Collection Workshop



# Downloading Latest CUMULVS Software Distribution

- Go To CUMULVS Web Site:

<http://www.csm.ornl.gov/cs/cumulvs.html>

or

<http://www.netlib.org/cumulvs> (orig version)

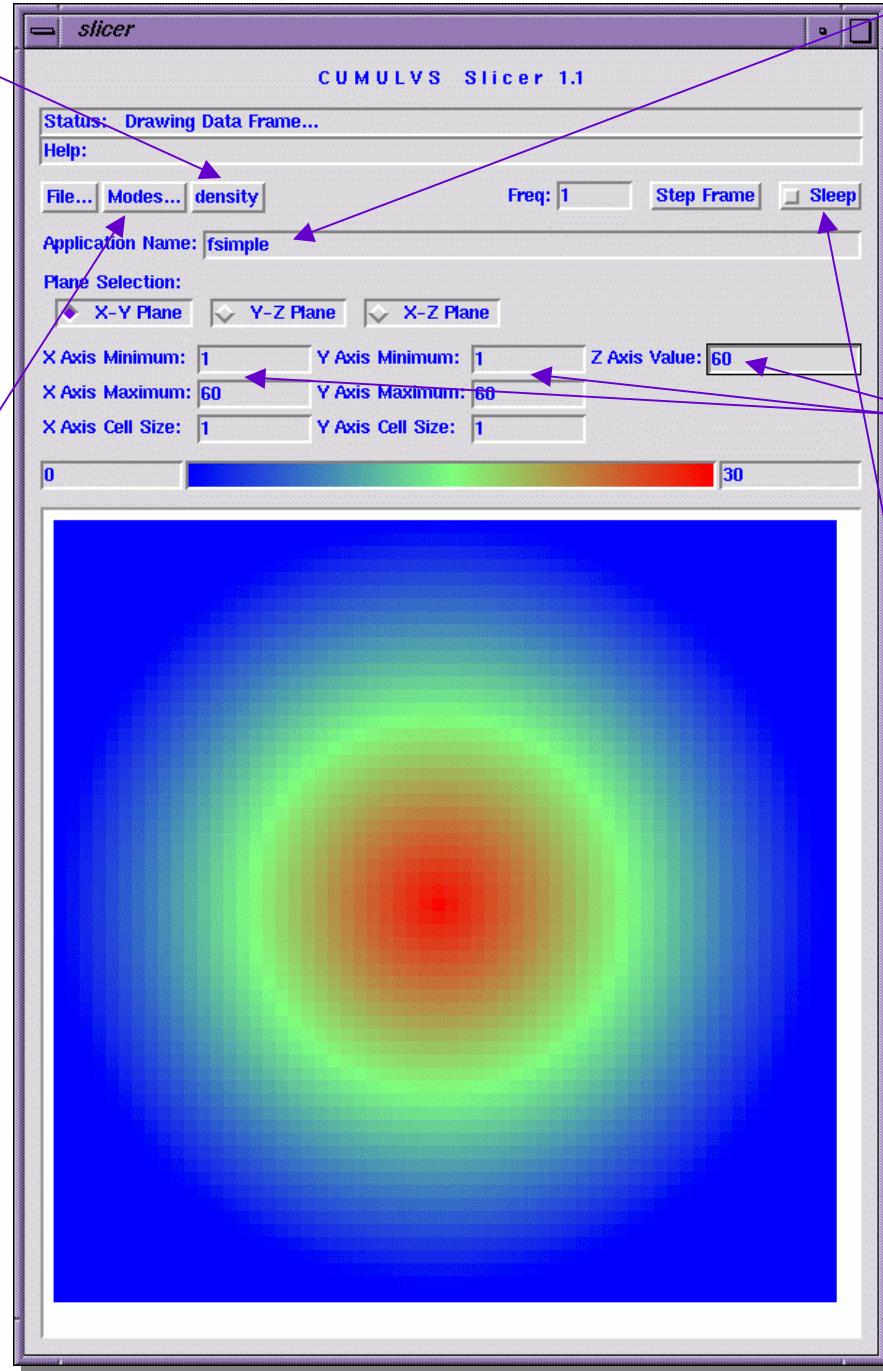
→ \*.tar.gz, \*.tar.Z, \*.tar.z.uu, \*.zip

# Slicer Interface

ORNL

2. Select  
Data Field  
Name:  
e.g. density

3. Select  
“Lazy” or “Auto”  
Color Map Mode.



1. Enter  
Application  
Name:  
e.g. fsimple

4. Set X,Y  
Region  
Boundaries,  
Z Value...

5. Toggle  
“Sleep”  
Button  
to Start...

# “Gif-O-Matic” Web Browser Viewer

The screenshot shows a window titled "CUMULVS Browser (GIF-O-Matic) 1.0". Inside the window, there is a large heatmap visualization with a color gradient from blue to red. Below the visualization, the text reads "Data Frame for Field "density" at 34896 MilliSecs (5 Min, 48 Secs, 696 MilliSecs)". At the top of the window is a menu bar with "File", "Edit", "View", "Go", "Bookmarks", "Tools", "Window", and "Help".

**Simple CGI/HTML Form Interface**

**Manual “Refresh”**

The interface includes several input fields and buttons:

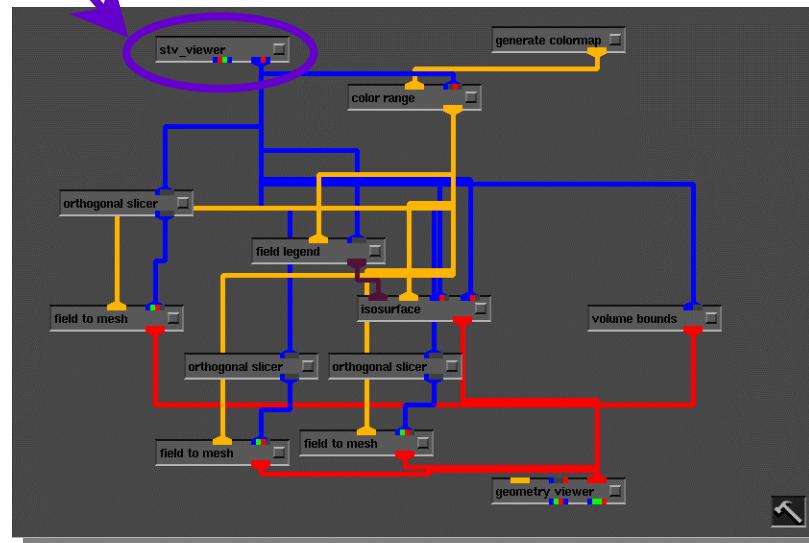
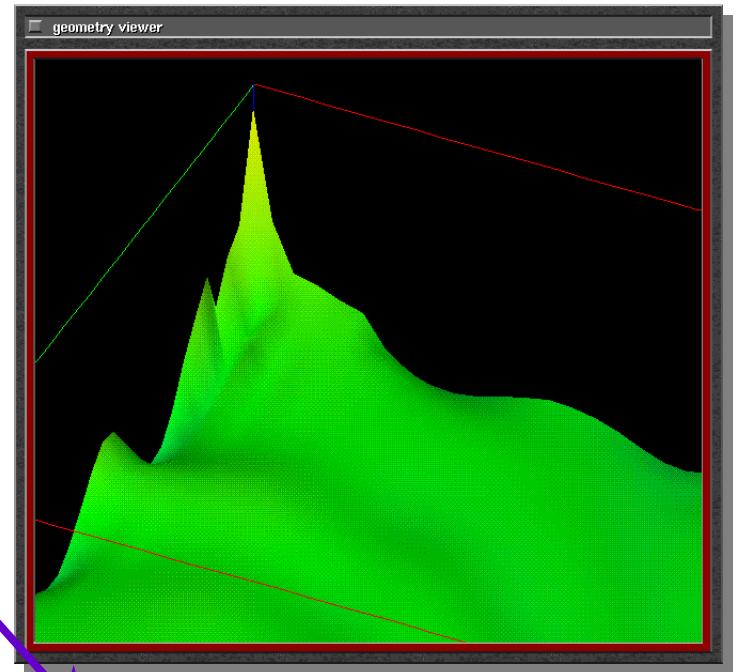
- Application Name:
- Data Field:
- Plane:
- Frame Freq:
- X Axis Minimum:
- Maximum:
- Cell Size:
- Y Axis Minimum:
- Maximum:
- Cell Size:
- Z Axis Value:
- Color Map Range: Minimum:  Maximum:  Mode:
- Change View:
- Refresh View:

At the bottom of the window, there is footer text: "CUMULVS Project - ORNL CSMD", "GIF-O-Matic and HTML-O-Matic Courtesy of SNI Distributed Computing Team", and "Research Sponsored by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC".

# CUMULVS Viewer for AVS5

**Visualization Settings**

Application Name	seismic
Data Field Name	pressure
Data Field Name List	
Value: pressure	
<not selected>	
pressure	
Frame Frequency	1
No Synch	
X Cell Size	1
X Min	0
X Max	45
Y Cell Size	1
Y Min	0
Y Max	45
Z Cell Size	1
Z Min	10
Z Max	60
Particle Cell Size	1
Min Data Value	-10
Max Data Value	10



**Sub Menus**

- Steering Parameters
- Visualization Settings

**Steering Parameter Request List**

Value: <not selected>
<not selected>

**Steering Parameter Release List**

Value: <not selected>
<not selected>
Shot X Coord
Shot Y Coord
Shot Z Coord

Shot X Coord	45
Shot Y Coord	55
Shot Z Coord	3
Thump	1

Only \*1\* Custom  
AVS Module...

# CUMULVS Build Environment

- Use Sample “Makefile.aimk” (examples/src/):

```
STV_ROOT = (where you installed CUMULVS)
STV_VIS = NONE
(STV_MP ~ No Longer Needed! ☺)
include $(STV_ROOT)/src/LINUX/Makeincl.stv
→ $(STV_CFLAGS), $(STV_LDFLAGS), $(STV_LIBS)...
→ Use "stvmk" script to build (invokes "make")
```

- Include CUMULVS Header Files (include/):

⇒ Defines Constants, Functions, Types

C:                   `#include <stv.h>`

Fortran:           `include 'fstv.h'`

# CUMULVS Initialization

(“Original”, Backwards Compatible Version)

- C: `stv_init( char *appname, int msgtag,  
int ntasks, int nodeid );`
- Fortran: `stvinit( appname, msgtag, ntasks, nodeid )`

where:

- appname ~ logical name of application
- msgtag ~ message tag to “reserve” for CUMULVS\*
- ntasks ~ number of tasks in application
- nodeid ~ task index of the caller

E.g.: `stv_init( "solver", 123, 32, 3 );`

\* For Backwards Compatibility with PVM 3.3 (Before Message Contexts...).

# CUMULVS MP-Friendly Initialization

(New Syntax, Includes Argc/Argv, e.g. for MPI\_Init()...☺)

- C: `stvInit( int *argc, char ***argv,  
char *appname, int msgtag,  
int ntasks, int nodeid );`

where:

- argc/argv ~ pointers to process command line arguments
- appname ~ logical name of application
- msgtag ~ message tag to “reserve” for CUMULVS\*
- ntasks ~ number of tasks in application
- nodeid ~ task index of the caller

E.g.: `stvInit( &argc, &argv, "solver", 123, 32, 3 );`

\* For Backwards Compatibility with PVM 3.3 (Before Message Contexts...).

# Distributed Data Decomposition

- C: `int decompId = stv_decompDefine( int dataDim,  
int *axisType, int *axisInfo1, int *axisInfo2,  
int *glb, int *gub, int prank, int *pshape );`
- Fortran: `stvfdecompdefine( dataDim, axisType, axisInfo1,  
axisInfo2, glb, gub, prank, pshape, decompId )`

where:

- **decompId** ~ integer decomp handle returned by CUMULVS
- **dataDim** ~ dimensionality of data decomposition
- **axisType** ~ decomp type identifier, for each axis
  - \* `stvBlock`, `stvCyclic`, `stvExplicit`, `stvCollapse`
- **axisInfo1,2** ~ specific decomposition details, per axis
  - \* E.g. `axisInfo1 == Block Size or Cycle Size`
  - \* E.g. `axisInfo1 == Explicit Lower Bound, axisInfo2 == Upper Bound`
  - \* Note: `axisInfo1,2` can be set to `stvDefaultInfo`

# Data Decomposition (cont.)

## Global Bounds and Pshape

- C: `stv_decompDefine( ...,`  
`int *glb, int *gub, int prank, int *pshape );`
- Fortran: `stvfdecompdefine( ..., glb, gub,`  
`prank, pshape )`

where:

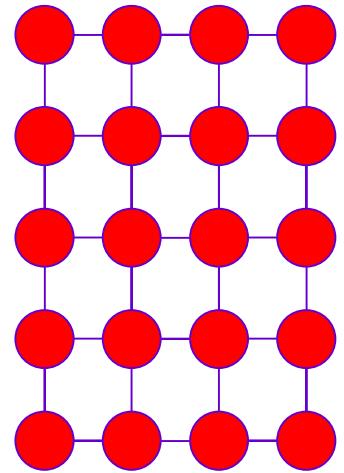
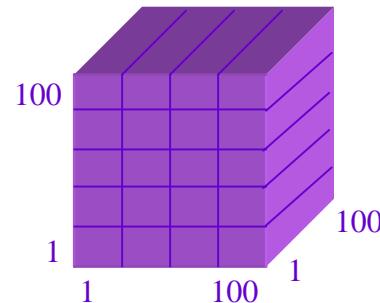
- **glb** ~ lower bounds of decomp in global coordinates
- **gub** ~ upper bounds of decomp in global coordinates
- **prank** ~ dimensionality of processor topology (pshape)
- **pshape** ~ logical processor organization/topology
  - \* number of processors per axis

# Example Decomposition #1

## 3D Standard Block Decomp on 2D Processor Array

```
int decompId, axisType[3], glb[3], gub[3], pshape[2];  
  
axisType[0] = axisType[1] = stvBlock;  
axisType[2] = stvCollapse;  
  
glb[0] = glb[1] = glb[2] = 1;  
gub[0] = gub[1] = gub[2] = 100;  
  
pshape[0] = 4;      /* "Standard" Block Size = 25 */  
pshape[1] = 5;      /* "Standard" Block Size = 20 */  
  
decompId = stv_decompDefine( 3,  
    axisType, stvDefaultInfo, stvDefaultInfo,  
    glb, gub, 2, pshape );
```

ORNL



# Example Decomposition #2

## 3D Block-Cyclic Decomp on 2D Processor Array

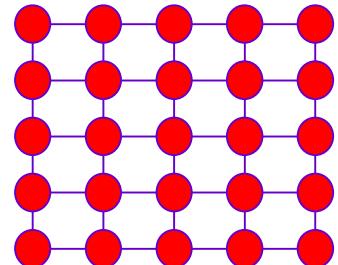
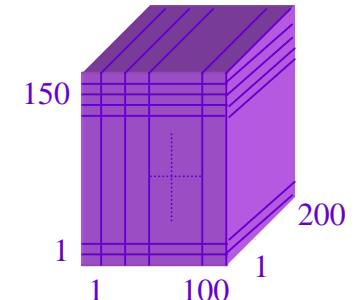
```
int axisType[3], axisInfo[3], glb[3], gub[3], pshape[2];

axisType[0] = stvBlock;    axisInfo[0] = 10; /* Block */
axisType[1] = stvCyclic;   axisInfo[1] = 3;  /* Cycle */
axisType[2] = stvCollapse;

glb[0] = glb[1] = glb[2] = 1;
gub[0] = 100;  gub[1] = 150; gub[2] = 200;

pshape[0] = 5;  pshape[1] = 5;

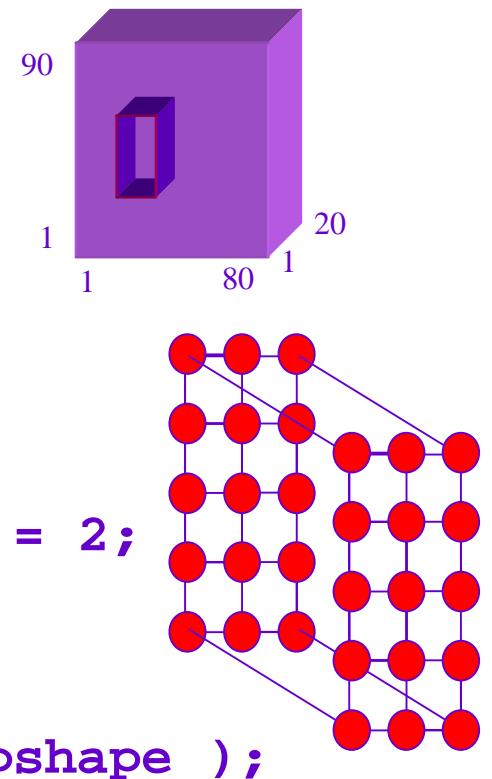
decompId = stv_decompDefine( 3, axisType,
                            axisInfo, stvDefaultInfo, glb, gub, 2, pshape );
```



# Example Decomposition #3

## 3D Explicit Decomp on 3D Processor Array

```
int axisType[3], axisInfo1[3], axisInfo2[3],... pshape[3];  
  
axisType[0] = axisType[1] = axisType[2] = stvExplicit;  
axisInfo1[0] = 11;  axisInfo2[0] = 17;  
axisInfo1[1] = 33;  axisInfo2[1] = 57;  
axisInfo1[2] = 1;   axisInfo2[2] = 7;  
  
glb[0] = glb[1] = glb[2] = 1;  
gub[0] = 80;  gub[1] = 90;  gub[2] = 20;  
  
pshape[0] = 3;  pshape[1] = 5;  pshape[2] = 2;  
  
decompId = stv_decompDefine( 3, axisType,  
    axisInfo1, axisInfo2, glb, gub, 3, pshape );
```



# Additional Explicit Regions

## For Extra Explicit Array Subregions Per Task

- C: `int stv_decompAddExplicitPatch( int decompId,  
int *lowerBounds, int *upperBounds );`
- Fortran: `stvfdecompaddexplicitpatch( decompId,  
lowerBounds, upperBounds, info )`

where:

- decompId ~ integer handle to CUMULVS decomp
- lowerBounds ~ lower bounds of subregion per axis  
(global coords)
- upperBounds ~ upper bound of subregion per axis  
(global coords)

# Example Decomposition #4

## 3D Local Array

```
int decompId, glb[3], gub[3];  
  
glb[0] = glb[1] = glb[2] = 0;  
gub[0] = gub[1] = gub[2] = 100;  
  
decompId = stv_decompDefine( 3,  
                             stvLocalArray, stvLocalArray, stvLocalArray,  
                             glb, gub, 1, stvLocalArray );
```

# Data Field Definition

- C: `int fieldId = stv_fieldDefine( STV_VALUE var,  
char *name, int decompId, int *arrayOffset,  
int *arrayDecl, int type, int *paddr, int aflag );`
- Fortran: `stvffielddefine( var, name, decompId,  
arrayOffset, arrayDecl, type, paddr, aflag, fieldId )`

where:

- **fieldId** ~ integer field handle returned by CUMULVS
- **var** ~ reference (pointer) to local data array storage
- **name** ~ logical name of data field
- **decompId** ~ handle to pre-defined decomposition template
- **type** ~ integer constant that defines data type
  - \* **stvByte**, **stvInt**, **stvFloat**, **stvDouble**, **stvLong**, **stvCplx**, **stvUint**, ...

# Data Field Definition (cont.)

## Local Data Field Allocation

- C: `int fieldId = stv_fieldDefine( ...,  
int *arrayOffset, int *arrayDecl, ... );`
- Fortran: `stvffielddefine( ..., arrayOffset, arrayDecl ... )`

where:

- **arrayOffset** ~ offset to “valid” data, along each axis
- **arrayDecl** ~ overall allocated array size, along each axis

Note: Size of actual “valid” data is determined in conjunction  
with data decomposition information...

# Data Field Definition (cont.)

## Processor “Address”

- C: `int fieldId = stv_fieldDefine( ..., int *paddr, ... );`
- Fortran: `stvffielddefine( ..., paddr, ... )`

where:

→ **paddr** ~ integer array containing the local task’s position  
in the overall processor topology (pshape)  
\* paddr index starts from 0!

(NOT the “physical” processor address!)

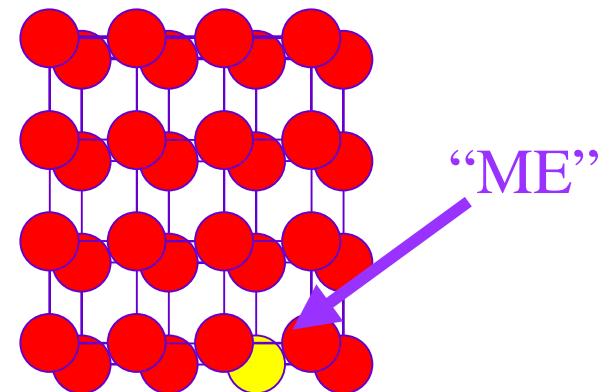
E.g. if Pshape = { 4, 4, 2 } and Prank = 3,

then Paddr = { 2, 3, 1 } represents:

the 3<sup>rd</sup> processor of 4 along axis 1

the 4<sup>th</sup> processor of 4 along axis 2

the 2<sup>nd</sup> processor of 2 along axis 3



# Data Field Definition (cont.)

## Access/Usage Flag

- C: `int fieldId = stv_fieldDefine( ..., int aflag );`
- Fortran: `stvffielddefine( ..., aflag, ... )`

where:

- **aflag** ~ integer flag to indicate external access to field:
  - \* **stvVisOnly** ~ access data field for visualization only
  - \* **stvCpOnly** ~ access data field for checkpointing only
  - \* **stvVisCp** ~ access data field for visualization *and* checkpointing

# Example Data Field #1

100x100x100 Float Array on Processor { 1, 2 } of Decomp  
(E.g., say, as part of 1000x1000x1000 Global Decomp)

```
float foo[100][100][100];
int fieldId, declare[3], paddr[2];

declare[0] = declare[1] = declare[2] = 100;

paddr[0] = 1;  paddr[1] = 2;

fieldId = stv_fieldDefine( foo, "Foo", decompId,
                           stvNoFieldOffsets, declare, stvFloat, paddr,
                           stvVisOnly );
```

## Example Data Field #2

80x80x60 Double Array on Processor { 2, 1, 3 } of Decomp  
(With 10 Element Neighbor Boundary in Local Allocation)

```
double bar[100][100][80];
int fieldId, offsets[3], declare[3], paddr[3];

offsets[0] = offsets[1] = offsets[2] = 10;

declare[0] = declare[1] = 100; declare[2] = 80;

paddr[0] = 2; paddr[1] = 1; paddr[2] = 3;

fieldId = stv_fieldDefine( bar, "Bar", decompId,
                           offsets, declare, stvDouble, paddr, stvVisCp );
```

# Particle Definition

## A Container for Particle Data Fields

- C: 

```
int particleId = stv_particleDefine( char *name,
                                         int dataDim, int *glb, int *gub, int ntasks,
                                         STV_GET_PARTICLE get_particle(),
                                         STV_MAKE_PARTICLE make_particle() );
```
- Fortran: 

```
stvfparticledefine( name, dataDim, glb, gub,
                           ntasks, get_particle, make_particle, particleId )
```

where:

- **particleId** ~ integer particle handle returned by CUMULVS
- **name** ~ logical name of particle wrapper
- **dataDim** ~ dimensionality of particle space
- **glb, gub** ~ global bounds of particle space
- **ntasks** ~ number of parallel tasks cooperating on particles

# Particle Definition (cont.)

## User-Defined Particle Functions

- C: `int particleId = stv_particleDefine( ...,`  
`STV_GET_PARTICLE get_particle(),`  
`STV_MAKE_PARTICLE make_particle() );`
- Fortran: `stvfparticledefine( ... , get_particle,`  
`make_particle, ... )`

where:

- **get\_particle( )** ~ user-defined accessor function
  - \* returns a user-defined handle to CUMULVS for the given particle
- **make\_particle( )** ~ user-defined constructor function
  - \* allow CUMULVS to create a new particle (checkpoint recovery)
  - (Can set make\_particle( ) to NULL if not checkpointing)

# Get a Particle for CUMULVS

## Particle Accessor Function

- C: `void get_particle( int index, STV_REGION region,  
STV_PARTICLE_ID *id, int *coords );`
- Fortran: `get_particle( index, region, id, coords )`

where:

- **index** ~ requested particle index
  - \* positive integer value, index set to **-1** for search reset
  - \* if particle not found for given index, should return **id** as NULL
- **region** ~ subregion of particle space to search
  - \* can use stv\_particle\_in\_region( ) utility function...
- **id** ~ user-defined particle handle returned by search
  - \* (`void *`) → anything the application wants to cast it to...
- **coords** ~ array of coordinates for returned particle

# Make a Particle for CUMULVS

## Particle Constructor Function – Checkpoint Recovery

- C: `void make_particle( int *coords,  
 STV_PARTICLE_ID *id );`
- Fortran: `make_particle( coords, id )`

where:

- **coords** ~ array of coordinates for requested particle creation
- **id** ~ user-defined particle handle returned for new particle
  - \* (`void *`) → anything the application wants to cast it to...

# Particle Field Definition

## Actual Data Fields Associated with a Particle

- C: `int pfieldId = stv_pfieldDefine( char *name,  
int particleId, int type, int nelems,  
STV_GET_PFIELD get_pfield(), STV_VALUE get_pfield_arg,  
STV_SET_PFIELD set_pfield(), STV_VALUE set_pfield_arg,  
int aflag );`
- Fortran: `stvfpfielddefine( name, particleId, type, nelems,  
get_pfield, get_pfield_arg, set_pfield, set_pfield_arg,  
aflag, pfieldId )`

where:

- **pfieldId** ~ returned integer particle field handle
- **name** ~ logical name of particle data field
- **particleId** ~ handle to encapsulating particle wrapper

# Particle Field Definition (cont.)

## Actual Data Fields Associated with a Particle

- C: `stv_pfieldDefine( ..., int type, int nelems, ...,  
int aflag );`
- Fortran: `stvfpfielddefine( ..., type, nelems, ...,  
aflag, ... )`

where:

- **type** ~ data type for particle field (**stvInt**, **stvFloat**, etc.)
- **nelems** ~ number of data elements
  - \* only simple 1-D arrays (vectors) are supported as particle fields
- **aflag** ~ same as for `stv_fieldDefine()`, access flag
  - \* **stvVisOnly**, **stvCpOnly**, or **stvVisCp**.

# Particle Field Definition (cont.)

## User-Defined Particle Field Functions

- C: `stv_pfieldDefine( ...,`  
`STV_GET_PFIELD get_pfield(), STV_VALUE get_pfield_arg,`  
`STV_SET_PFIELD set_pfield(), STV_VALUE set_pfield_arg ... );`
- Fortran: `stvfpfielddefine( ...,`  
`get_pfield, get_pfield_arg,`  
`set_pfield, set_pfield_arg, ... )`

where:

- **get\_pfield( ) / get\_pfield\_arg** ~ user-defined accessor/arg
  - \* return reference to actual data storage
  - \* extra user-defined argument, (void \*) for vectored get\_pfield( ) impl
- **set\_pfield( ) / set\_pfield\_arg** ~ user-defined constructor/arg
  - \* for checkpoint recovery, set the values of given particle field array
  - \* extra user-defined argument, (void \*) for vectored set\_pfield( ) impl

# Get a Particle Field for CUMULVS

## Particle Field Accessor Function

- C: `void get_pfield( STV_PARTICLE_ID id,  
STV_VALUE user_data, STV_VALUE *data );`
- Fortran: `get_pfield( id, user_data, data );`

where:

- **id** ~ user-defined particle handle
  - \* as returned by previous `get_particle()` call
- **user\_data** ~ arbitrary user-defined argument
  - \* (`void *`) value for vectored implementation of `get_pfield()`
- **data** ~ reference to actual data storage

E.g. `if ( user_data == "density" )  
*data = ((mystruct) id)->density;`

# Set a Particle Field for CUMULVS

## Particle Field Constructor Function – Checkpoint Recovery

- C: `void set_pfield( STV_PARTICLE_ID id,  
STV_VALUE user_data, STV_VALUE data );`
- Fortran: `set_pfield( id, user_data, data );`

where:

- **id** ~ user-defined particle handle
  - \* as returned by previous `get_particle( )` call
- **user\_data** ~ arbitrary user-defined argument
  - \* `(void *)` value for vectored implementation of `set_pfield( )`
- **data** ~ reference to new data values to set

E.g. `if ( user_data == "density" )  
((mystruct) id)->density = data;`

# Steering Parameter Definition

- C: `int paramId = stv_paramDefine( char *name,  
STV_VALUE var, int type, int aflag );`
- Fortran: `stvfparamdefine( name, var, type, aflag,  
paramId )`

where:

- **paramId** ~ returned integer steering parameter handle
- **name** ~ logical name of steering parameter
- **var** ~ reference to actual steering parameter storage
- **type** ~ data type of steering parameter (**stvInt**, **stvFloat**, ...)
- **aflag** ~ external access flag, same as before
  - \* **stvVisOnly**, **stvCpOnly** or **stvVisCp**

# Steering Parameter Definition

## Vector Parameters

- C: 

```
int paramId = stv_vparamDefine( char *name,
    STV_VALUE *vars, char **pnames, int *types, int num,
    int aflag );
```
- Fortran: (no Fortran equivalent yet)

where:

- **paramId** ~ returned integer steering vector handle
- **name** ~ logical name of steering vector
- **vars** ~ array of references to steering element storage
- **pnames** ~ array of logical names for steering elements
- **types** ~ array of steering element data types
  - \* OR with **stvIndex** for “indexed” steering vector (**stvInt** | **stvIndex**)
- **aflag** ~ external access flag (**stvVisOnly**, **stvCpOnly**, **stvVisCp**)

# Steering Parameter Updates...

- C: `int changed = stv_isParamChanged( int paramId );`
- Fortran: `stvfisparamchanged( paramId, changed )`

where:

- **paramId** ~ integer steering parameter handle
- **changed** ~ status of steering parameter
  - \*  $> 0$  indicates an update has occurred
  - \*  $= 0$  indicates no change to parameter value
  - \*  $< 0$  indicates an error condition (bad parameter id, etc)

# CUMULVS Coupling Interface

- C: `int couplerId = stv_couple_fields( int fieldId,  
char *appStr, char *fieldName,  
int dstFreq, int srcFreq,  
int dstNoSynch, int srcNoSynch, int block );`
- Fortran: `stvcouplefields( fieldId, appStr, fieldName,  
dstFreq, srcFreq, dstNoSynch, srcNoSynch,  
block, couplerId )`

where:

- **fieldId** ~ locally defined data field (destination)
- **appStr/fieldName** ~ location of remote data field (source)
- **dstFreq/srcFreq** ~ frequency of coupling at source/destination
- **dstNoSynch/srcNoSynch** ~ synchronization requirements
- **block** ~ determines blocking/non-blocking coupling attempt
- **couplerId** ~ handle for manipulating coupler connection

# CUMULVS Coupling Interface

## Disconnection

- C: `int stv_couple_finished( int couplerId );`
- Fortran: `stvcouplefinished( couplerId )`

where:

→ **couplerId** ~ handle for coupler to disconnect

# Passing Control to CUMULVS

- After Data Fields and Parameters Are Defined
  - ⇒ Single Periodic Call to Pass Control to CUMULVS
  - ⇒ Once Per Iteration? Between Compute Phases?

C: `int params_changed = stv_sendReadyData(`  
    `int update_field_times );`

Fortran: `stvfsendreadydata( update_field_times,`  
        `params_changed )`

where:

→ **params\_changed** ~ return code that indicates whether  
any steering parameter values have been updated

\* > 0 an update has occurred, = 0 no changes to any values

→ **update\_field\_times** ~ flag to control data field “times”

\* in general, just use **stvSendDefault...** ☺

# CUMULVS Checkpoint Initialization

- C: `int cp_restart = stv_cpInit( char *aout_name,  
int notify_tag, int *ntasks );`
- Fortran: `stvfcpinit( aout_name, notify_tag, ntasks,  
cp_restart )`

where:

- **cp\_restart** ~ status value, indicates restart / failure recovery
  - \* use to circumvent normal application startup and data initialization!!
- **aout\_name** ~ name of executable file to spawn for restart
- **notify\_tag** ~ message code for notify messages (PVM only)
- **ntasks** ~ number of tasks in application
  - \* in a restart condition, this value can be set by CUMULVS!

Note: `stv_cpInit()` must be called **after** `stv_init()`!

# Alternate Checkpoint Restart Check

- C: `int cp_restart = stv_isCpRestart();`
- Fortran: `stvfiscprestart( cp_restart )`

where:

→ **cp\_restart** ~ status value, indicates restart/failure recovery  
\* use to circumvent normal application startup and data initialization!!

# CUMULVS Option Interface...

## Checkpointing Option

- One Option of Many, Part of Larger Interface...
- Invoke **Before** You Call `stv_cpInit()`! ☺

```
/* Set Recovery Option to Rollback */
stv_setopt( stvDefault, stvOptCpRecovery,
            stvOptCpRollback );
```

or

```
/* Set Recovery Option to Restart */
stv_setopt( stvDefault, stvOptCpRecovery,
            stvOptCpRestart );
```

---

Same in Fortran:

```
stvfsetopt( STVDEFAULT, STVOPTCPRECOVERY,
            STVOPTCPRESTART ) . . .
```

# Checkpoint Collection

- Trigger Actual Data Collection from Local Task
- Invoke When Parallel Data/State Consistent
  - ⇒ Highly Non-Trivial in General!! (Chandy/Lamport)
  - ⇒ Straightforward for Most Iterative/Phased Applications
    - E.g. Save Checkpoint at Beginning or End of Main Loop

- C: `int status = stv_checkpoint();`
- Fortran: `stvfcheckpoint( status )`

where:

→ **status** < 0 indicates system failure

Control Overhead Using Checkpointing Frequency:

```
if ( !(i++ % 100) ) stv_checkpoint();
```

# Restoring Data From A Checkpoint

- After Restart Condition Has Been Indicated...
  - Can Be Invoked Incrementally to Bootstrap Data
    - ⇒ Interlace with Data Field/Parameter Definitions...
    - ⇒ Automatically Loads Data Directly Into User Memory for All Defined Data Fields and Parameters
      - C: `int remaining = stv_loadFromCP();`
      - Fortran: `stvfloadfromcp( remaining )`
- where:
- **remaining** ~ number of remaining data fields or parameters to define/restore

# Example Instrumentation

## Multi-Stage Restart from a Checkpoint...

```
/* Check Restart Status... (Returned By stv_cpInit( ) ) */

/* Load Program Variables From Checkpoint Data */
if ( restart )  stv_loadFromCP( );

/* Allocate "indices" Vector Using "delta" for Size */
ix = (int *) malloc( ( 100.0 / delta ) * sizeof(int) );

/* Define "indices" Parameter for CUMULVS */
stv_paramDefine( "indices", ix[], stvInt, stvVisCp );

/* Load Newly-Defined Program Var ix[] from Ckpt Data */
if ( restart )  stv_loadFromCP( );
else  user_init_data( );
```

# Oh Yeah, One Last Routine... ☺

## Finished Checkpointing

```
 . . .

/* Tell CUMULVS to Stop Checkpoint Recovery. */
/* (So Task Can Exit Normally & R.I.P. :-) */
stv_cpFinished( );

exit( );
}
```

# CUMULVS Interface Summary

- Interact with Scientific Simulations
  - ⇒ Dynamically Attach Multiple Visualization Front-Ends
  - ⇒ Steer Model & Algorithm Parameters On-The-Fly
  - ⇒ Couple Disparate Simulation Models
  - ⇒ Automatic Heterogeneous Fault Recovery & Migration

<http://www.csm.ornl.gov/cs/cumulvs.html>

Email: [cumulvs@msr.csm.ornl.gov](mailto:cumulvs@msr.csm.ornl.gov)