RedGRID: Related Works

Lyon, october 2003.

Aurélien Esnard

EPSN project (ACI-GRID PPL02-03)

INRIA Futurs (ScAlApplix project) & LaBRI (UMR CNRS 5800)

351, cours de la Libération, F-33405 Talence, France.

Outline

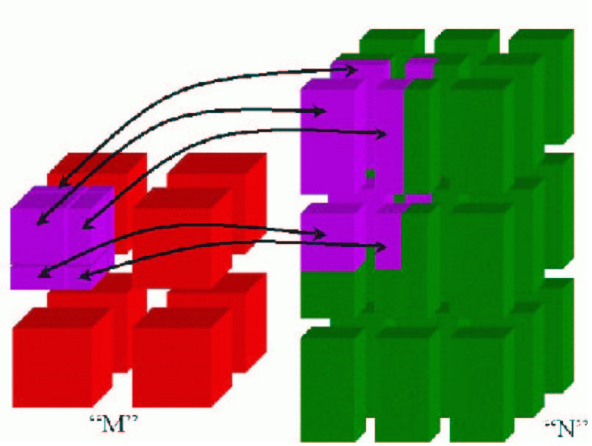
1. Introduction
2. CUMULVS
3. PAWS
4. Meta-Chaos
5. CCA MxN
6. Conclusion

Introduction

Challenges

Coupling parallel applications and sharing parallel data structures between these codes

- **efficient parallel transfer** $\rightarrow$ avoid **serialization** bottlenecks
- using different parallel layout strategies (**redistribution**)
- **heterogeneous** data transfer (languages & systems)
- provide several **synchronization** strategies
- **dynamic** coupling of applications, at any time during their execution



Separation of Layout & Actual Data

Layout (data distribution)

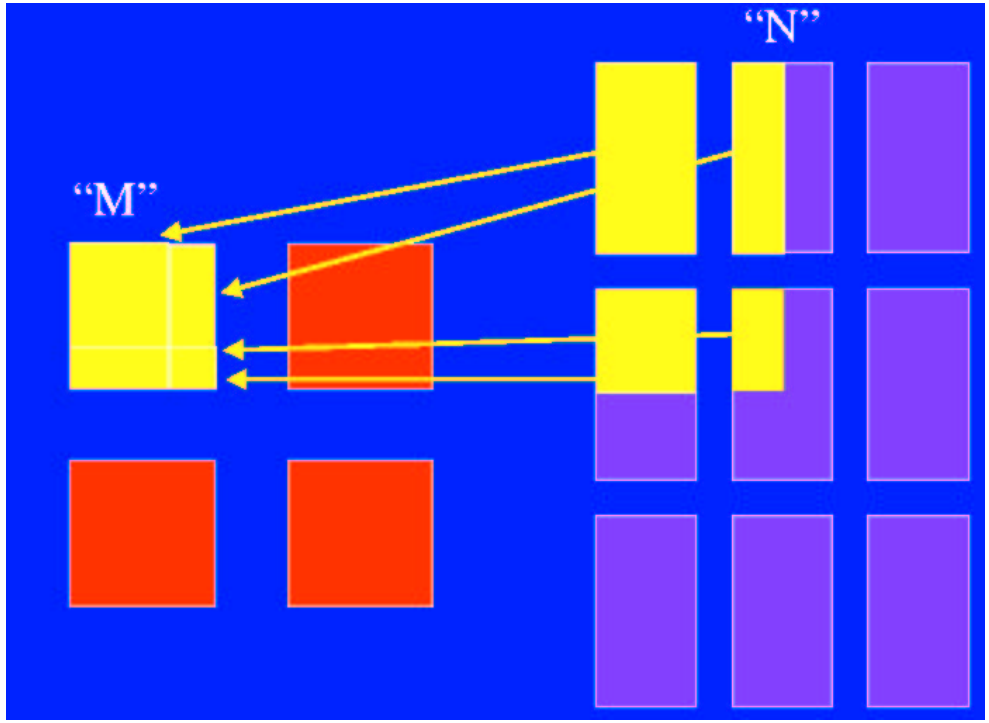
- specification of global domain (dimensions, index space)
- specification of distribution pattern
- specification of logical process topology

Actual Data (local allocation)

- specification of the data type
- allocation of storage space (address)
- placement of process in logical process topology
- link data object to layout (including alignment of data within layout)

Communication Schedules

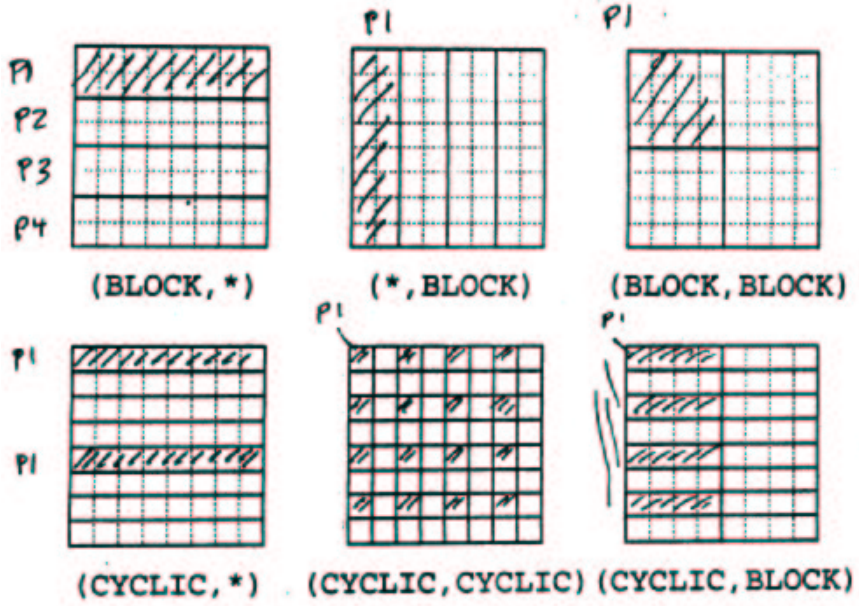
Communication Schedules (just) built from layouts on both sides



HPF Decomposition

Axis decomposition

- **block:** one block per processor ($size = N/p$)
- **cyclic:** many blocks of same size per processor, cyclically assigned
- **explicit:** user-defined decomposition (index ranges per processor)
- **collapsed:** not decomposed along the axis

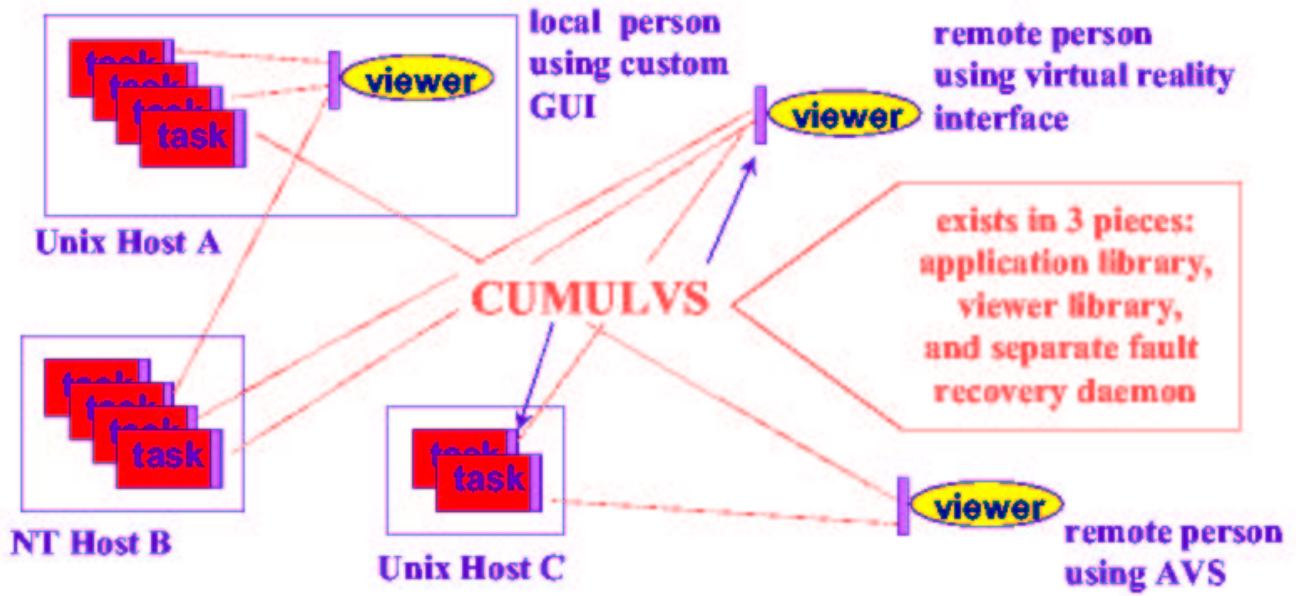


CUMULVS

Overview

CUMULVS: Collaborative Infrastructure for Interacting With Scientifics Simulations (ORNL)

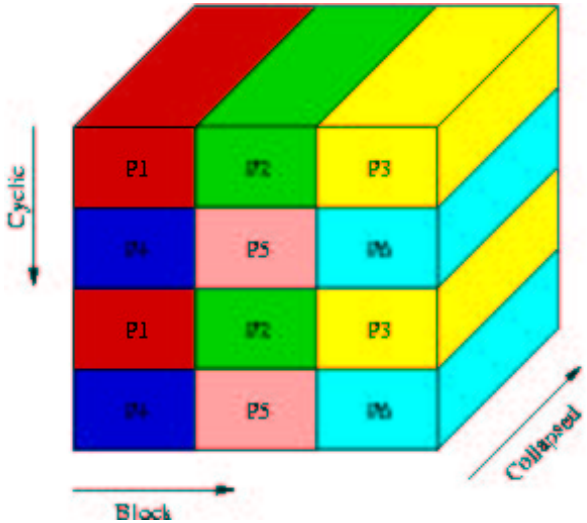
- remote run-time visualization, computational steering, checkpointing
- coupling parallel simulation and single-process visualizer (M by 1)
- communication based on PVM
- C and Fortran API
- data fields & particle fields (explicit coordinates)
- HPF-like distribution pattern

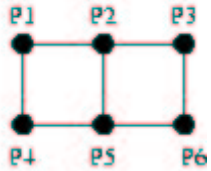


Data Decomposition

```
int decompId = stv_decompDefine(int dataDim, int *axisType, int *axisInfo1, int *axisInfo2,
     int *glb, int *gub, int prank, int *pshape);
```

- global domain dimension and index space (*dataDim, glb, gub*)
- block, cyclic, explicit or collapsed HPF decomposition per axis (*axisType, axisInfo1, axisInfo2*)
- processor topology (*prank, pshape*)



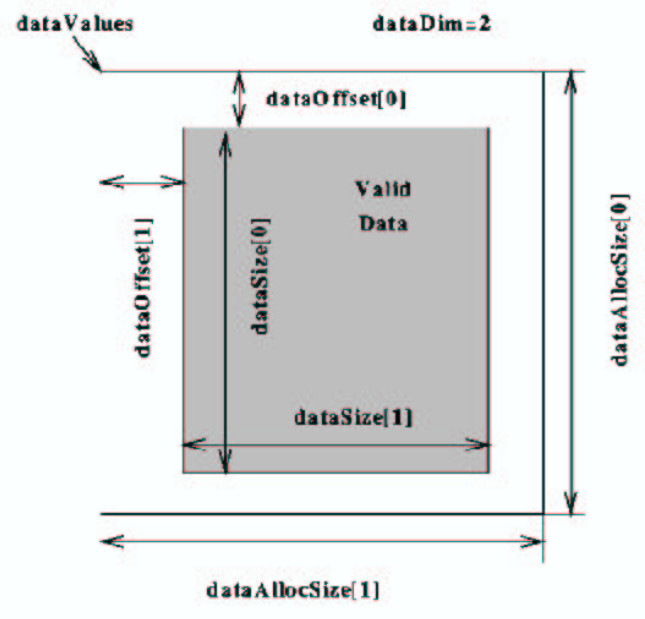


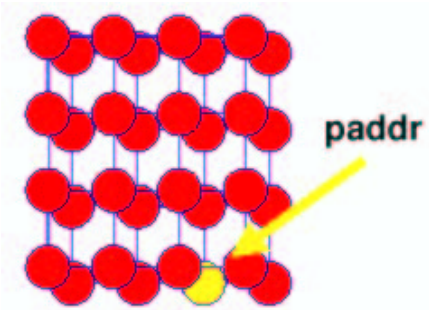
Process logical topology

Data Fields

```
int fieldId = stv_fieldDefine(void *var, char *name, int decompId, int *arrayOffset,
    int *arrayDecl, int type, int *paddr, int aflag);
```

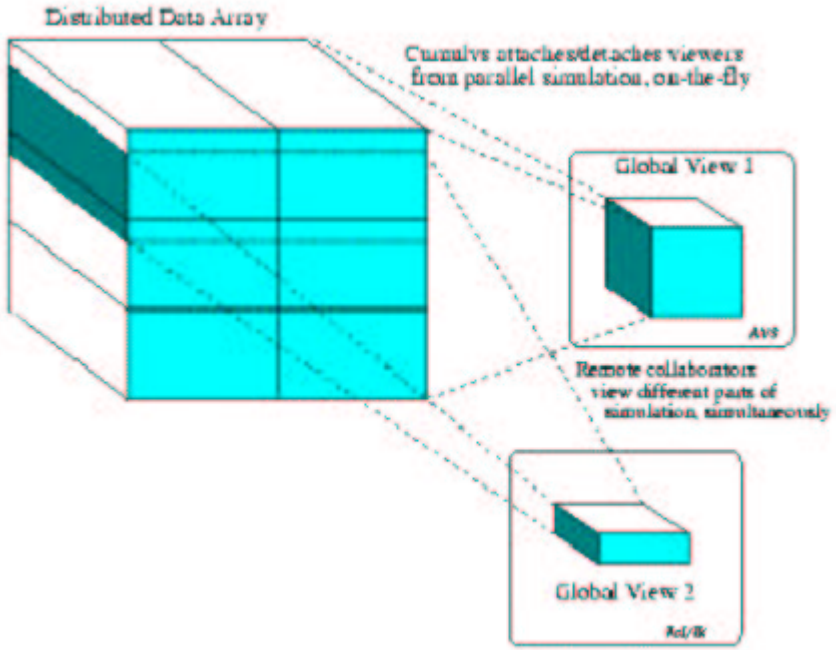
- local data array storage (*var*)
- decomposition handle (*decompId*)
- local data organization (*arrayOffset, arrayDecl*)
- type (*byte, int, float, double, long...*)
- logical processor address (*paddr*)
- remote access permissions (*aflag*)





View Fields

- multiple different views
- collection of data fields (view field group)
- different storage order (row/column major)
- different target type (e.g. double $\rightarrow$ int)
- portion of the computational space which is collected for viewing
- vis. region
  - set of upper and lower coordinate bounds for each dimension (subset)
  - cell size for each dimension (striding)



Communication Infrastructure

- simulations/applications uses MPI, PVM...
- CUMULVS based on PVM
  - message-passing
  - task & ressource management
  - independant spawns $\rightarrow$ dynamic attachment of viewers
  - heterogeneity (XDR)
  - fault-tolerance, etc.
- periodically pass control to CUMULVS (`stv_sendToFE()`)
- synchronous calls, frequency

CUMULVS Classical Example

Simulation

```
// initialize myapp for CUMULVS
stv_init("myapp",msgtag,nproc,nodeid);
// define data decomposition
did =  stv_decompDefine(...);
// define data field
fid =  stv_fieldDefine("myfield",...);
// define steering parameter
pid =  stv_paramDefine("myparam",...);

// main work loop
do {
  // execute computation
  done = work(&timestep);
  // pass control to CUMULVS
  nparams =  stv_sendToFE();
} while(!done);
```

Viewer

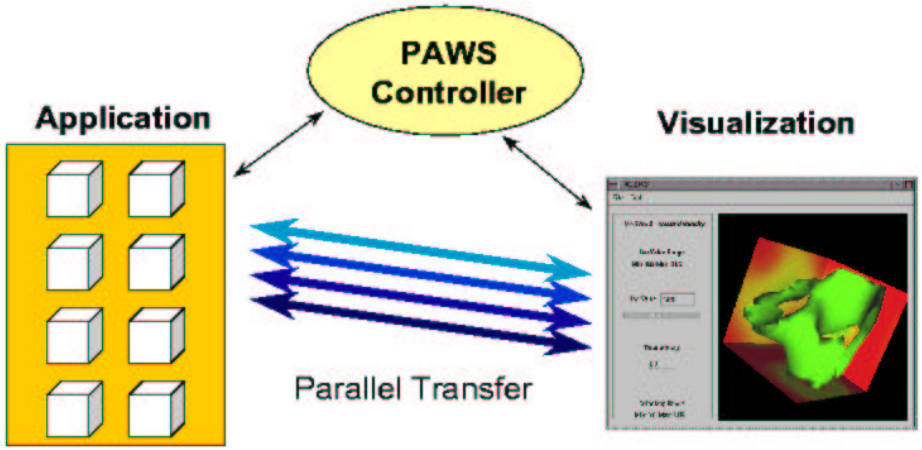
```
// attach CUMULVS viewer to myapp
stv_viewer_init(&viewer,mytid,"myapp",...);
// get field name description
vf =  stv_get_view_field_name("myfield",viewer...);
vf->selected = stvTrue;
vf->view_type = vf->field->type;
vf->view_storage_order = vf->field->storage_order;
// set vis region bounds and cell sizes
STV_REGION_MIN(visregion,i) =  vf->field->decomp->glb[i];
STV_REGION_MAX(visregion,i) = vf->field->decomp->gub[i];
STV_CELL_OF_DIM(viscell,i) = 1;
// request field from myapp
vfg =  stv_viewer_submit_field_request(viewer,visregion,
            viscell,freq,...);

// main vis loop
do {
  // collect data frame from myapp
  cc =  stv_viewer_receive_frame(&vfg,viewer,...);
  // do something here...
  // acknowledgment
  stv_viewer_send_XON(vfg,viewer);
} while(!done);
```

PAWS

Overview

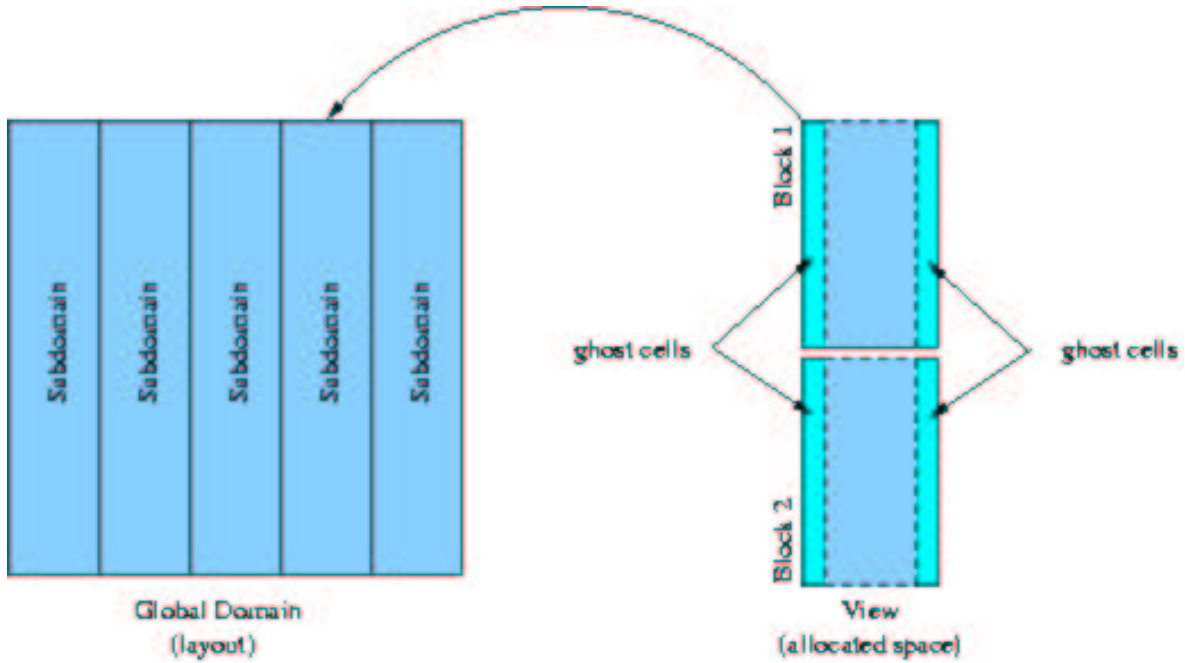
PAWS: Parallel Application Workspace (LANL)

- coupling of parallel applications using parallel communication chanels
- a component-like model (port)
- different number of processes (M by N)
- C, C++ and F77 API
- single scalar & rectilinear parallel data
- be extensible to new, user-defined parallel distribution strategies

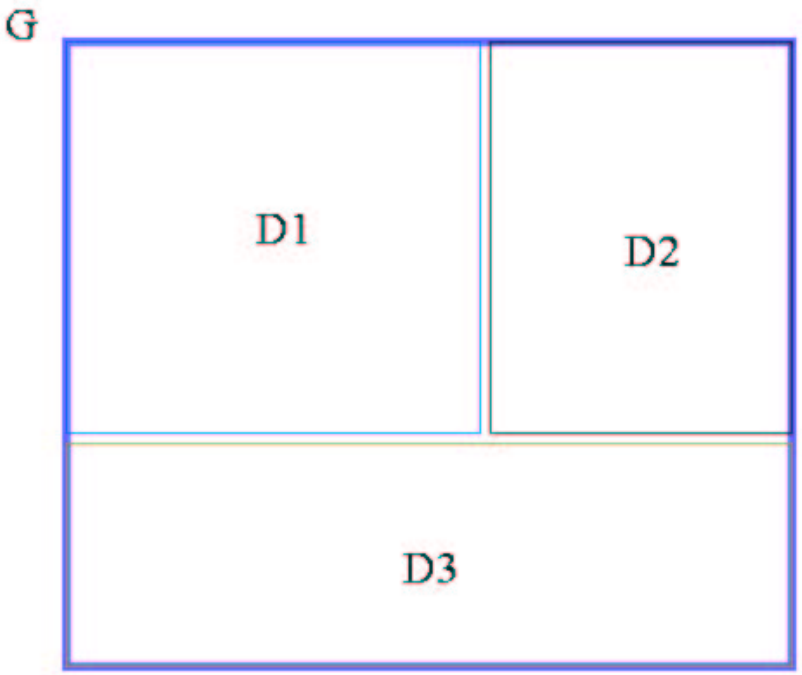


Parallel Data Representation

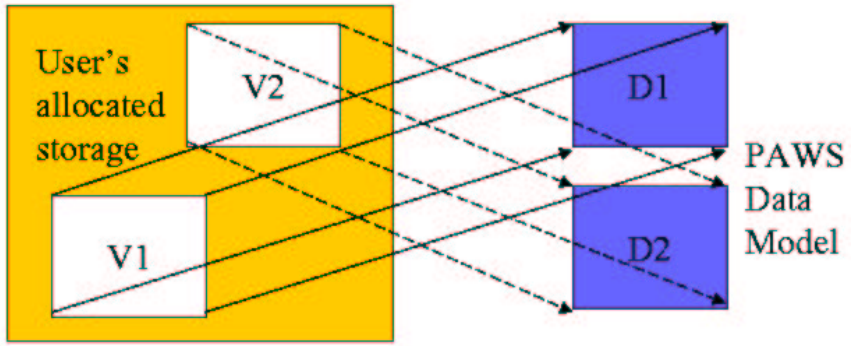
- any distributed dense rectilinear data
- *nomenclature:* **representation** (layout), **view** (actual data)
- representation: global domain G and a list of local subdomains $\{D_i\}$
- view: a list of blocks $\{(first, last, stride)_i\}$



Illustration



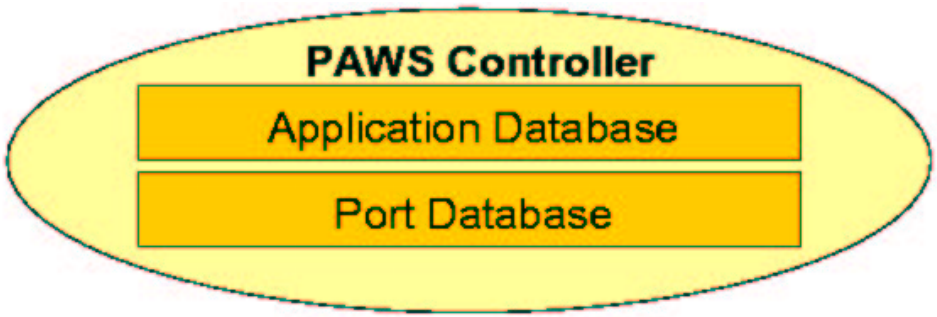
representation (layout)



view (actual data)

Communication Infrastructure

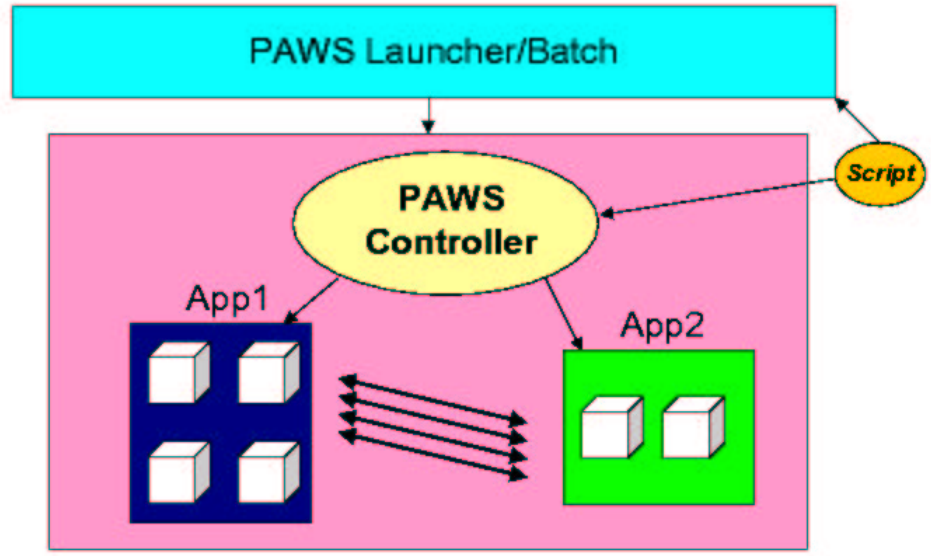
- **channel abstraction** $\rightarrow$ port are connected, not data
- PAWS based on NEXUS (Globus communication layer)
- independant of the application's parallel communication mechanism
- sender application & receiver application connected through PAWS Controller
- fully synchronous, partially synchronous, fully asynchronous connnections
- PAWS Controller (repository for applications, data and connections)
- query controller databases through Tcl scripting interface or PAWS API



PAWS Controller & Applications

Scenario

1. launch PAWS Controller and applications (by hand or through PAWS Launcher)
2. App1 & App2 register to PAWS Controller
3. App1 & App2 register data structure (ports)
4. establish connections (through the controller script interface or PAWS API)
5. PAWS Controller computes communication schedules (from both layouts)
6. send/receive data
7. disconnects ports or app. from PAWS Controller



PAWS Classical Example

Sender

```
// register with PAWS
paws_initialize(argc,argv);
// create ports
rep = paws_representation(wholeDom,myDom,myRank);
A = paws_view_port("A",rep,PAWS_OUT,PAWS_DISTRIB);
// allocate data and create views
int * data = (int*)malloc(mySize*sizeof(int));
view = paws_view(PAWS_INTEGER,PAWS_ROW);
paws_add_view_block(view,data,myAllocDom,myViewDom);
// ready
paws_ready();

// main work loop
do {
  // execute computation
  done = work(&timestep);
  // send data (synchronous)
  paws_send(A,view);
} while(!done);

paws_finalize();
```

Receiver

```
// register with PAWS
paws_initialize(argc,argv);
// create ports
rep = paws_representation(wholeDom,myDom,myRank);
B = paws_view_port("B",rep,PAWS_IN,PAWS_DISTRIB);
// allocate data and create views
int * data = (int*)malloc(mySize*sizeof(int));
view = paws_view(PAWS_INTEGER,PAWS_ROW);
paws_add_view_block(view,data,myAllocDom,myViewDom);
// ready
paws_ready();

// main work loop
do {
  // receive data (synchronous)
  paws_view_receive(B,view);
  // do something here...
} while(!done);

paws_finalize();
```

Meta-Chaos

Meta-Chaos (U. Maryland)

- interoperability between different data parallel libraries (libX & libY)
- multiple libraries can exchange data in the same data parallel program or between different parallel programs (Fig. 1,2)
- different number of processes, different data organizations
- not only for distributed arrays!
- **meta-library** built from a set of interface functions that every data parallel library must export (location of distributed data)
- disadvantage: interfaces must be provided by the original library that implements the data structures

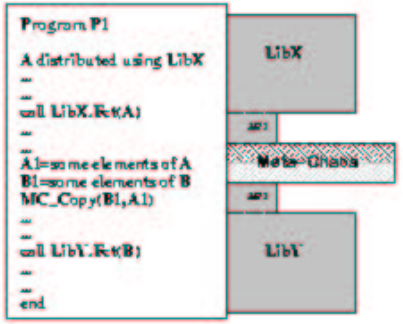
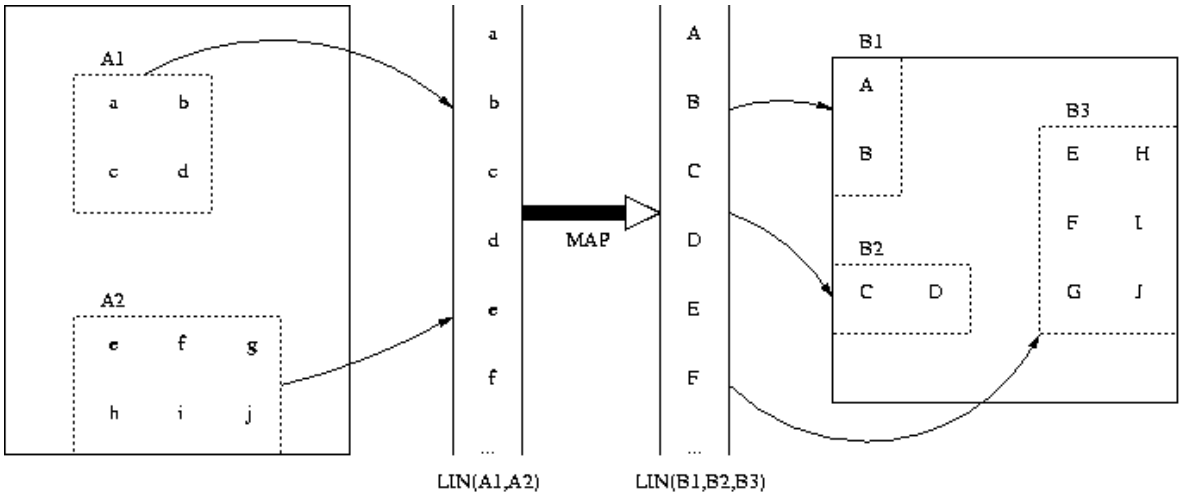


Figure 1. Communicating between two libraries within the same program

Figure 2. Communicating between libraries in two different programs

Linearization

- **set of regions** (region type = block of elements / set of array indices)
- copy elements from the source set to the target set
- **linearization**: one to one implicit mapping from the source to the destination (total ordering for the elements)
- linearization “flattens” the data structure into one dimensional linear array



More Exotic Data Structures!

- linearization concept independant of the original data structure
- **no limited to multidimensional arrays!**
- distributed aggregates, including pointer-based structures (*trees, graphs, unstructured meshes*)
- e.g. depth-first linearization for trees
- cf. Chaos library for irregular distribution...

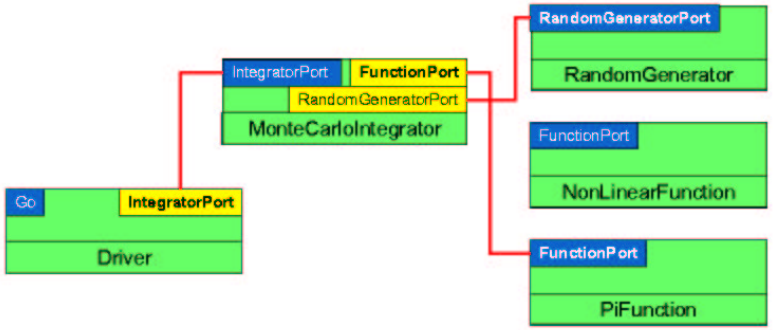
Communication Infrastructure

- communication based on MPI or PVM
- messages are aggregated
  $\rightarrow$ one message is sent between each source and each destination processor
- steps needed to copy data distributed using one library to data distributed using another library
  1. specify the elements to be copied (sent) from the 1st data structure, distributed by libX
  2. specify the elements to be copied (received) into the 2nd data structure, distributed by libY
  3. specify the correspondance (*mapping*) between the elements to be sent and the elements to be received
  4. build a communication schedule, by computing the locations (processors and local addresses) of the elements in the two distributed data structures
  5. perform the communication using the schedule produced in step 4

CCA MxN

CCA (Common Component Architecture)

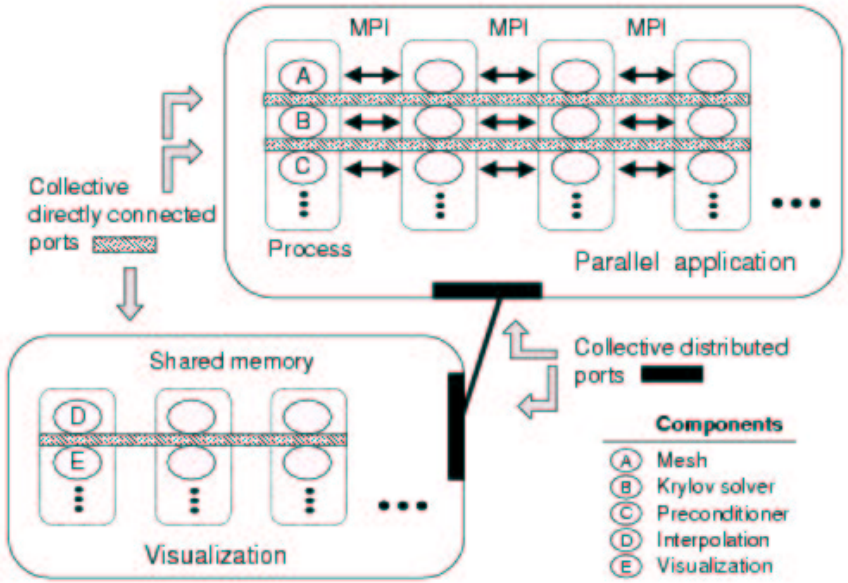
- specification for a component environment (specifically designed for HPC)
- concepts: **components**, **ports**, and **frameworks**
- components interacts through well-defined interfaces (ports)
- advantages: reusable functionality, composability, well-defined interfaces, etc.



- a component may provide a port (implement the interface)
- a component may use that port (call methods in that port)
- framework holds the components and compose them into applications

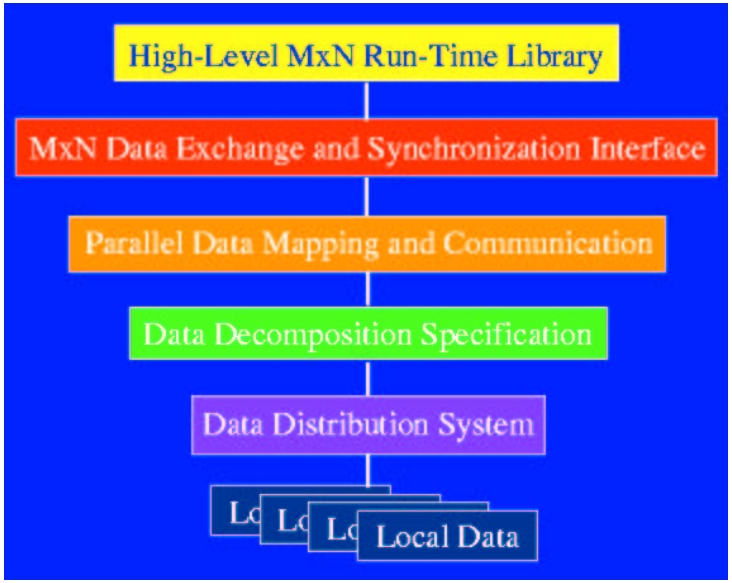
CCA for HPC, Collective Interactions

- language interoperability (Babel, SIDL)
- **parallel component:** collaboration of multiple threads or processes that logically represents one computation (user-defined communication within a parallel component)
- **collective ports:** collective invocation on all processes or on a proper subset (CCA extension)
- direct connections maintains local performances (a virtual function call)



CCA MxN Working Group

- specification interface to allow CCA components to identify and exchange data elements among parallel, decomposed data objects
- encapsulate existing technologies: CUMULVS, PAWS, Meta-Chaos, PADRE...
- data decompositions, communication schedules, connection & synchronization protocols...
- MxN Prototypes at SC01 (cf. CUMULVS/MxN, PAWS/MxN)
- No one tool fully implements MxN!



MxN Interface Keypoints (2002/04)

- data registration (local information and distributed data decomposition)

```
All A: registerData(dataA,sync,access,handleA);
All B: registerData(dataB,sync,access,handleB);
```

- communication schedules (map together two parallel entities)

```
All A: createCommSched(dataB.decomp,dataA.decomp,handleCS);
```

- MxN connections (one-shot or periodic connections)

```
All A: makeConnection(handleB,handleA,handleCS,syncB,syncA,freqB,freqA,handleC);
```

- request parallel data transfer

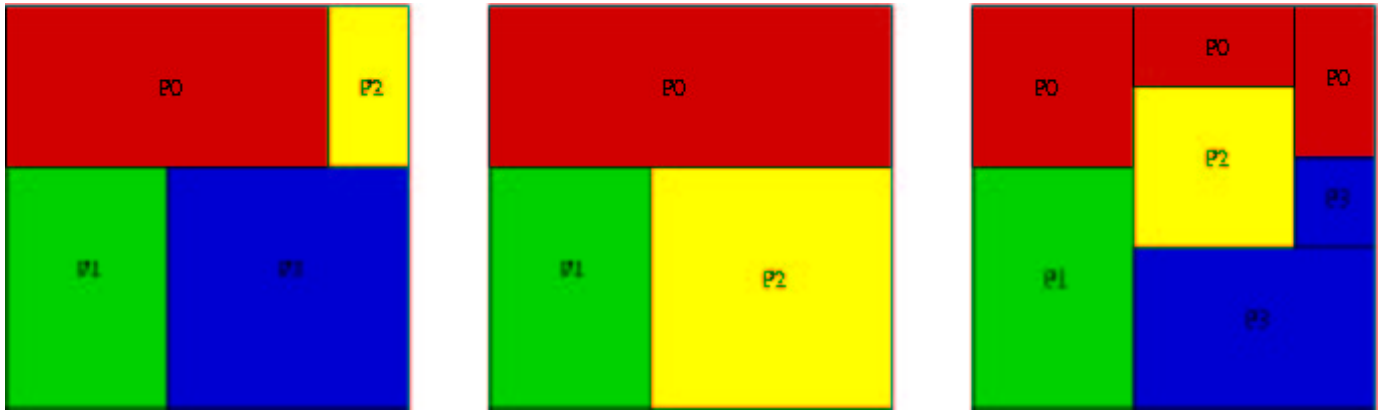
```
All A:
  requestTransfer(handleC,handleT);
  ...
  waitTransfert(handleT);
```

- data ready (parallel consistency of data, activate local transfer)

```
All A:
  dataReady(handleA,readwrite,flag);

All B:
  dataReadyBegin(handleB,readwrite,flag);
  ...
  dataReadyEnd(handleB);
```

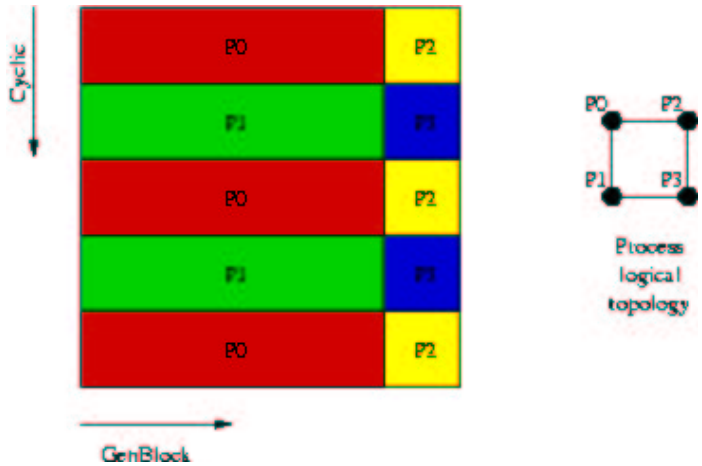
MxN Data Field Proposal (2002/06)

- distributed multidimensionnal, rectangular, arrays
- *nomenclature:* **template** (layout) & **descriptor** (actual data)
- decomposition types: *collapsed, block (regular & cyclic),* **genblock, implicit,** *explicit*
- several kinds of alignment: *identity, offset or general (HPF alignment)*

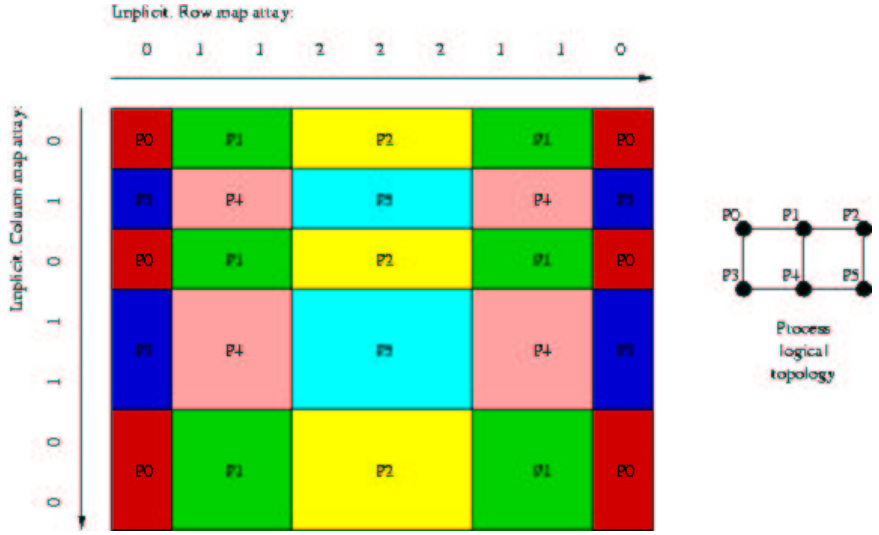


explicit

MxN Data Field Proposal (2002/06)

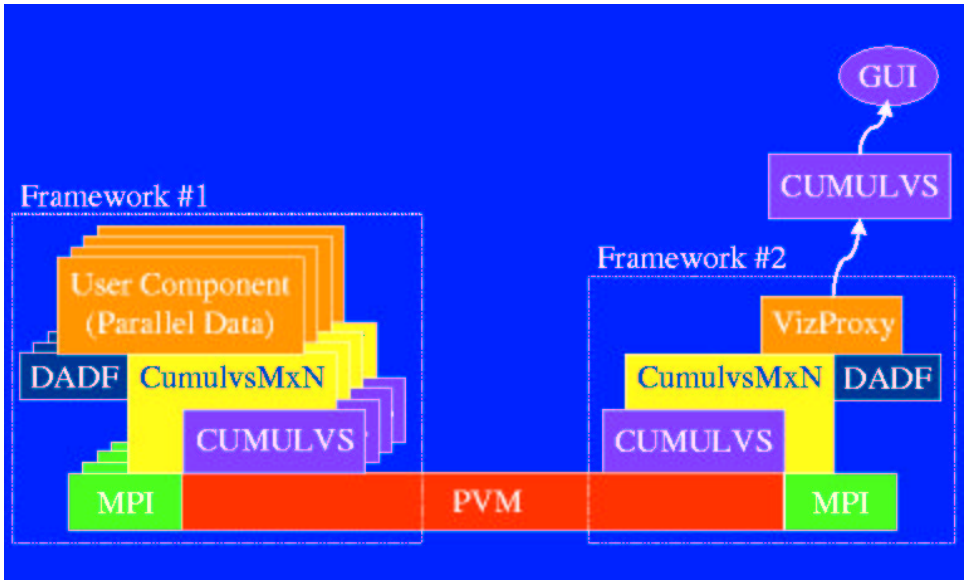


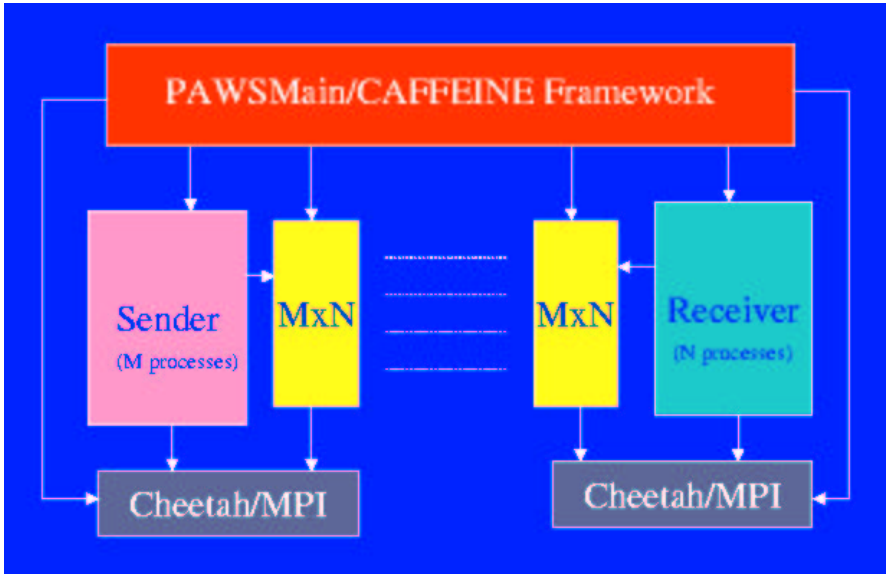
blocks of arbitrary sizes (one per process)



arbitrary mapping of elements to processes

MxN Prototypes





CUMULVS

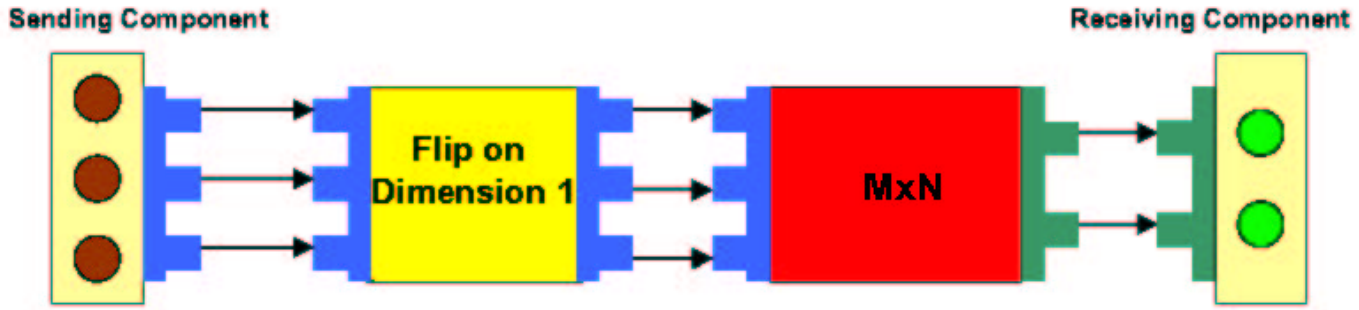
- 2 frameworks solution
- DADF component (Data Array Desc. Factory)
- CumuvsMxN component
  $\rightarrow$ for Viz (Mx1) and for App. (MxN)
- VizProxy component (Visualization)

PAWS

- single framework solution
- built on PAWS data description
- 2 MxN components $\rightarrow$ MxN_S for sender and MxN_R for receiver
- 3rd party control component

PAWS Translation Components

- beyond MxN component for data field
- translation components (different number of processes, different distributions, dimensional flip, column/row major, striding, compostion/decomposition)
- composing translation components



Conclusion

Summary

Library	Coupling	Objects	Comm.	Sync.
CUMULVS	Mx1	rectilinear array, particles	PVM	sync.
PAWS	MxN	rectilinear array	Nexus/MPI	sync/async
Meta-Chaos	MxN	irregular data	MPI/PVM	?
CCA MxN	MxN	rectilinear array	*framework*	sync/async
RedGRID	MxN	?	CORBA	?

"Goodies"

- other data structures (particles, unstructured meshes, trees...)
- more translations (not only MxN)
- advanced synchronization mechanism (coupling with visualization)
- extraction of correlated data
- extraction of a subset (sliding)

THE END.