

DARPA Data Re-organization Interface Effort  
San Diego, February 2, 1999

Attendance:

Darwin Ammala, MSTI	<a href="mailto:dammala@mpi-softtech.com">dammala@mpi-softtech.com</a>
Clayborne Taylor Jr., MSTI	<a href="mailto:cdtaylor@mpi-softtech.com">cdtaylor@mpi-softtech.com</a>
Steve Paavola, Sky	<a href="mailto:paavola@sky.com">paavola@sky.com</a>
Shane Hebert, MSU	<a href="mailto:shane@erc.msstate.edu">shane@erc.msstate.edu</a>
Ken Cain, MITRE	<a href="mailto:kcain@mitre.org">kcain@mitre.org</a>
Arkady Kanevsky, MITRE	<a href="mailto:arkady@mitre.org">arkady@mitre.org</a>
Dennis Cattel, SPAWAR (host)	<a href="mailto:dennis@spawar.navy.mil">dennis@spawar.navy.mil</a>
Nathan Doss, LM/GES	<a href="mailto:nathan.e.doss@lmco.com">nathan.e.doss@lmco.com</a>
James Lebak, MIT/LL	<a href="mailto:jlebak@ll.mit.edu">jlebak@ll.mit.edu</a>
Jon Greene, Mercury	<a href="mailto:greene@mc.com">greene@mc.com</a>
Richard Massary, NAWCAD	<a href="mailto:massaryrj@navair.navy.mil">massaryrj@navair.navy.mil</a>
Robert Grim, NAWCAD	<a href="mailto:rob@sai19.nawcad.navy.mil">rob@sai19.nawcad.navy.mil</a>
Randall Judd, SSC-50	<a href="mailto:judd@spawar.navy.mil">judd@spawar.navy.mil</a>

The goal for the meeting is to finish the discussion on API pre-requisite ideas, and begin to develop function calls.

### **Data Re-organization Steps**

Data → Logical Arrangement → Physical Arrangement → Processing Resource  
(These last 3 steps must be scalable)  
→ Decompose data into a logical arrangement  
→ Derive a physical arrangement from the logical  
→ Map the physical arrangement onto the processing resource.

### **Figure 1: Basic steps in data decomposition**

#### **Datatypes: Data representation**

For datatypes, we could adopt MPI/RT's versions: short, int, long, float, etc., the implementation will be free to extend them.

We don't want to specify a size for each type, as this would not be uniform for all architectures.

For global data re-organizations, do we want a group or communicator? - describe a collective entity for all participants?

#### **Partitioning:**

Partitioning objects should describe what is needed for the computation; e.g., I need a contiguous row. We must account for 2 consumers who want different parts of the data.

In a process set, we should agree on the global data object, and how it is to be partitioned. A strategy for dividing data needs to be Euclidean, slices must be parallel to a dimension. No striding (every nth element). Jon's PART\_XY discusses slicing along dimensions, you 'can' do so, vs. 'must'

#### **Use of 4-tuples, and reasonable default partitioning.**

We could allow both, specific or a reasonable default. A user can specify, or we can pick the default for them.

### Default, and specific API.

There could be multiple APIs for partitioning, e.g., `part_x_no_overlap`.

Support a *sizeof(record)*. A record is arbitrary in type and size.

It may have memory holes, e.g. 8 byte aligned and a *struct* of size 6 or 7.

### Local Processing

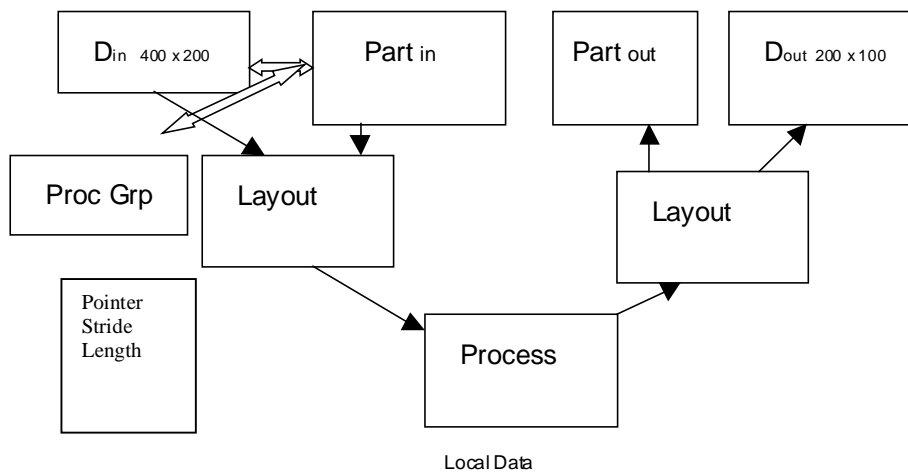
We need  $n$  (from  $D$ ) 4-tuples for the partition object.

On the sending side of  $D$ , we have pieces coming from somewhere, .."this is what I am providing (...)"

There is a notion of ownership. On sending side, overlap may not make sense. On input, the data set is partitioned disjointedly (otherwise conflicts on writes would arise).

On the receiving side, not every partition might be removed, this could be ok.

(index of start, index of end, ...) ?



**Figure 2: Local Processing**

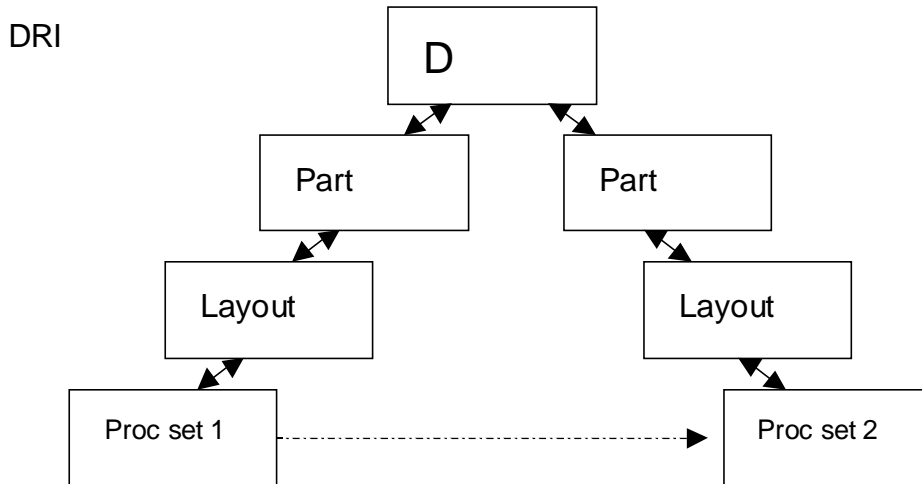
Take  $D$ , everyone owns a piece.

- the upper and lower pictures of  $D$  (Figures 1,2) partition, layout, process/transform.

-

Strategy of partitioning: transform  $D$ , input, process, output  $D'$ .. In local case, we might be decimating a corner turn into 4 pieces. e.g., a  $400 \times 200$  becomes 4 of  $200 \times 100$ . There should be NO overlaps on output to create to global  $D'$ .

### How to Get from the Partition to Layout Object



**Figure 3: Global View – Use of DRI**

### Partitioning Parameters

#### PART\_XY

1. Accept the default partitioning
2. Set some constraints
3. Fully specify the partitioning

### Figure 4: Partition in 2 dimensions – options

In partition specification:

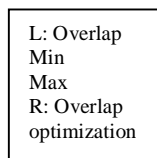
Part\_xy is how to slice the data in D. It can also mean, what part is independent.

Partition class, describes a min, max for a dimension, and which dimension is to be contiguous.

Additionally, the left and right offsets are specified. We have a constraint portion, and a requirement portion as stages. This would allow a failure to be detected earlier.

Min and max are constraints, and overlap is the requirement.

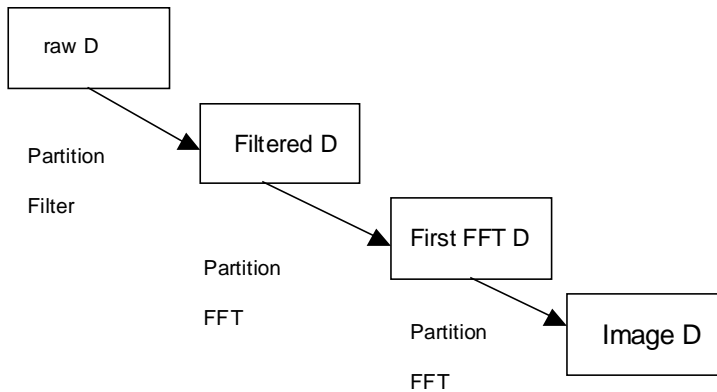
For each dimension, a min and max number of elements can be specified. To say exactly how many, set max and min equal. E.g., if we want 50 elements set both min and max to 50.



**Figure 5: Overlap Parameters – See also in Figure 2**

Partitioning should support heterogeneity since we might have specialized Resource Constrained processors which might have a filter in firmware, and a 2<sup>nd</sup> in software.

Raw Data --> Filter --> FFT --> Turn --> FFT -->

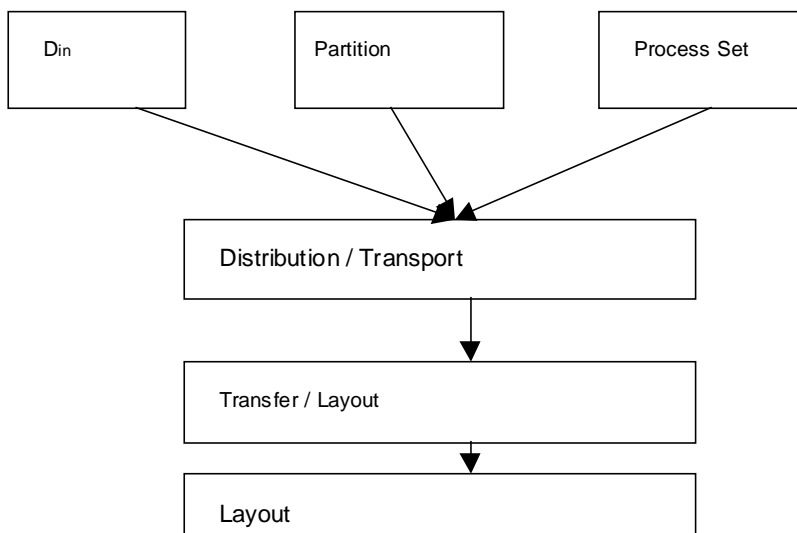


**Figure 6: Two stage FFT : One FFT could be in firmware, the Data, partition D, and process set object is transformed to a transfer object. The distribution object must have D, partition in, process set in, partition out, process set out. The transformation object is made of 2 distribution objects. Assumption, D is constant through a transform**

### Building of the Transfer and Layout Objects

The transfer object is made by collective call to all processes, this object has a handle, which is used in transfer operations.

It is also possible to give named objects. Knowing all distribution aspects, the sender and receiver could rendezvous at some point and transfer the data.

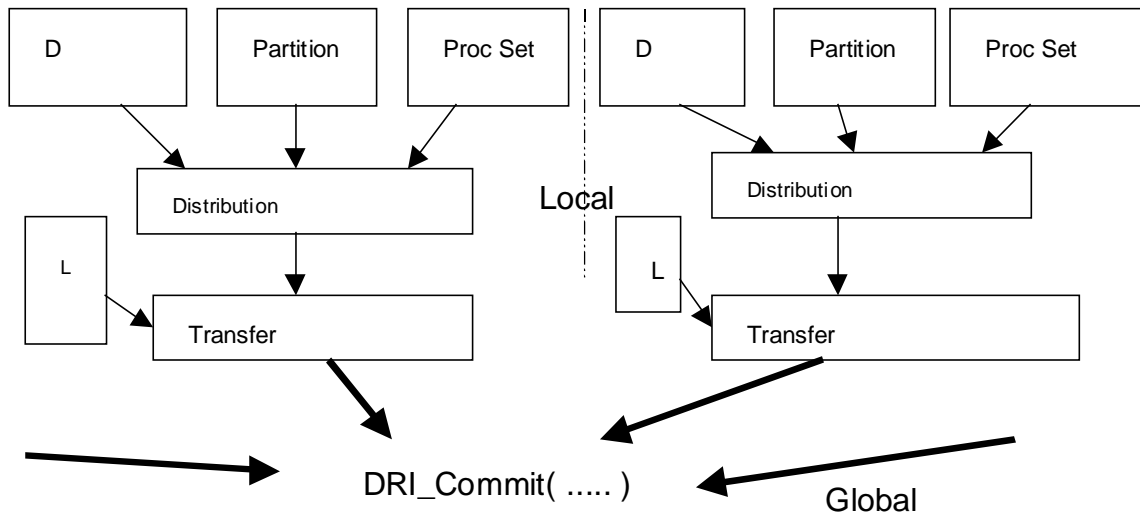


**Figure 7: Collective nature of building Transfer and Layout Objects**

A single process only needs to know its role (sender), a distribution is defined, we say we are the source, we can create a name to the transfer.

The receiver does the same thing. They rendezvous. Thus, a producer doesn't need to specify the source and destination distributions. A process is never both producer and consumer in the same step. It can be done in one operation a producer/consumer needn't do both steps in place. There can a buffer for each role. In a clique corner turn all are both producer and consumer, D is the same. Figure 7 shows the creation of the objects from the input side, output works analogously.

### Commit Operation



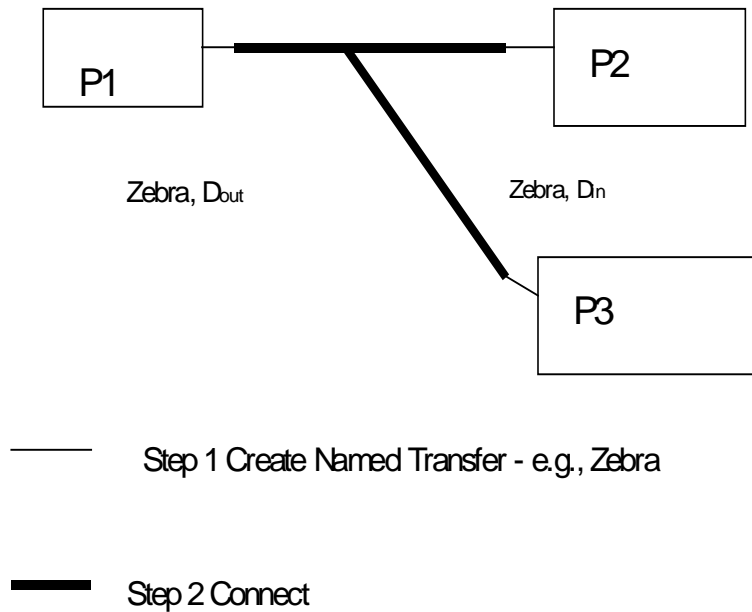
**Figure 8: Local and Global aspects of Commit operation**

The user must create buffers in the commit stage of an operation. D and partition are local. The process set has not been addressed. The distribute is scoped to a process set. After the collective call, all checks have been made to stay within the definitions of D Part, Process Set. The Distribution object creation is synchronizing, we must avoid deadlock.

Named transform stage, we must be collective across both ends of the transfer. With a name given to the transfer object, more than 1 destination can synchronize. However, we are postponing error detection for the commit operation.

Outside of the transform call, we may not know if the operation will succeed, or be legal since the transform object is local.

## Named Transfers



**Figure 9: Named Transfers**

The name represents a communication; at commit time the name and resources can be known, P1 is the source of transform zebra. Processes p2, p3 consume it, and p2/3 may not know of each other.

What if p3 requests call before p1 is ready? We have blocking and non-blocking calls.

At the end of the commit, we get objects for all handles that were set up in earlier stages.

What about 1 sided push, pull, or 2 sided?

Push - p1 does a **do** data goes to both p2,3

Pull from p3 implies p2 gets the data too, if only 1 'zebra' name is used for transfer.

During transfer, p1 has blocking calls (collective d) thus no push/pull.

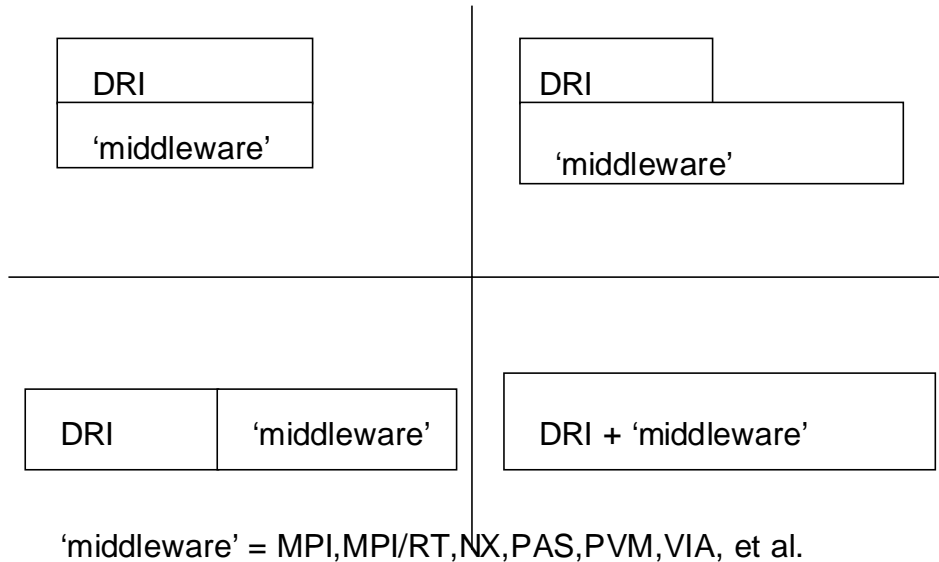
In tight resource situations, we can either write a smaller D, to deal with less buffer space, or back off as an error.

In a large operation, such as a 17k x 96k reduction and corner turn in parallel. The corner turn and computation are overlapped. You could set smaller than 17k pieces. The API won't implicitly support this type of operation. An Option would be to double buffer the input or turned object.

Question: how static is the buffer assignment? We would not want to have to rebuild everything. We also don't want to re-invent all of RT's buffer constructs - pools etc.

## The Nature and Intention of the DRI API

Assumption: API will change, as it is implemented with different libraries. E.g., MPI, and RT, and PAS will all look different. DRI could also exist on its own. The goal is to be conceptually portable vs. literally. DIR should be viewed as a concept unifier versus and end unto itself.



**Figure 10: Where does DRI fit in?**

In early binding: we allocate buffers, and link the buffer into distribution object. The buffer could be an array. No formal queuing implied, simply cycles through data.

Buffers are allocated locally, in the global case it is allocated at commit.

The application owns input/output buffers, also inter-process buffers at the interfaces would have other buffers. - used in Go call. Call buffer is in the layout, which the user gives to the interface. Local buffers are bound early, and at runtime the go call buffers are allocated. It marries go, pack/unpack.

Within the transfer, each side must issue the **go** call.

Good, fast memory is given early to the interface before runtime. Size is provided by the user. The memory is owned by the application. The transfer for local to global forces a copy. DRI must figure optimal way to move data – lest we have a *commit cart and horse problem*. You won't know until commit time on the receive side how big the buffers are. The distribution side does know their size. If no transpose, you may not need local buffers but you won't know this until commit time. We would like to allocate memory early, but we don't know how much to allocate that early.

The forced copy will hurt performance. Is the local implementation 'smart' enough to avoid local copy. You could look at the address and test. You are likely to have to do the copy anyway.

Can we use double buffers on both sides to avoid the copy? The buffers would be associated with the distribution object. Both allocated by user. One buffer is temporary, while the other is active. It's implied that the active buffer is sent by the **go** call. This would imply that the same address would be used for the transfer - hence early binding. Sometimes we need more than two buffers. You may not know how many buffers will be needed until commit time. In clique operations you'd have input, output and temp (3) buffers. Might be able to use the temp buffer for other actions so this may not be as wasteful.

**Addressing the Commit cart and horse:** We know who is interconnected - we can make the 'netlist'. This whole netlist issue can be resolved with respect to the buffer space allocation since all of the important

information could be known early. Commit may still have to be at the end. Commit would set up DMA engines.

**Dennis' proposal:** Before you make calls to create the transform object, make a register for named connection calls e.g., Make (use zebra). All those must complete before transform\_create is done. Real buffers must be known by both ends in order to avoid copies.

Distribute call: you get the size of buffer, you bind to object. At commit time all is known. You then decide you need a big temp buffer. Internally tries to create. User might provide specialized knowledge on which transfers are not overlapped.

Ken: data endpoint buffers, must be created internally. It must be acceptable for user to do a malloc etc. we need this to be portable.

Layout object simplest, packed xyz orientation.

### Summary:

6 Objects:

1. D: data shape object

2. Partition object, of 4-tuples per dimension. We haven't resolved this fully

We don't yet know details they are euclidian, not sparse. We will have guidance on partitioning.

3. Process set object an array of handles for processes in the communication.

4. These three are fed into a Distribution object creation call (collective over the process set). Error checking will be done here. Named vs. ordered. Named implies asynchronous. It must be a synchronized call. Even if named it must be blocking - hence decision is to have it be ordered over the process set. We know our part of D after this, our 4-tuples are then also known.

5. A layout object is a packed permutation of dimensions.

6.. A Transfer object. This can be asynchronous across all process sets. But it should be synchronous in local group..

Within a process set, all data belongs to a process set, no data belongs to more than one process in the set.

Do we want to push burden of partitioning up to the user to ensure scalability. They could write a program to do the mapping. Partition should be precisely defined on the source side specify such as to not imply overlap. Communication steps are the proper place to handle pack/unpack You can chose not to send overlapped data, but then you need to jump over it.

Convenience functions create D' as function of D. How do we enter to API. Also do convenience functions from :P to P'

Ned to agree on low level partition attributes to support them directly.

### The go call

We now can write down a set of functions, elaborate on objects, the go and Do calls, synchronization, moving of buffers, etc.

GO: is it blocking and synchronous. User declares buffers range cycle through buffers double, triple just cycle through them - then they can be used by a receiver who sends etc.

Steve: we may not know which piece of global data we are generating at a given time. We are one of the processes in the set,

The interface is static, a change implies a new transfer object. We might want to consider an offset change. A future Go could have extra arguments to handle the more difficult cases as we approach them.

If Go returned a structure with descriptors and status, you could learn of partitioning info  
Question of buffer usage. We may not want Go to determine this.

We should write up a simple case of this.

#### **API in brief:**

Several families of functions will be defined in the DRI\_API.  
Create family – does the first portion of the committal of resources  
Query family – enquiry of status  
Destroy family – return resources to the system  
Init family – sets up initial state

The Create family is outlined here, specifications for the others will follow.  
The format for the specification will be language independent.

DRI\_gdo\_create (n-dimensions, DRI\_datatype, sizes[], &gdo\_h)

// Create a generalized data object.

IN n-dimensions is number of dimensions,  
IN datatype is fixed for sizes array of sizes,  
IN sizes is the array of sizes  
INOUT gdo\_h gdo object handle.

DRI\_part\_create (n-dimensions, DRI\_part\_tuples[], &part\_h)

// Create a partition object

IN n-dimensions is the number of dimensions  
IN DRI\_part\_tuples is the array of partition 4-tuples, one tuple for each dimension  
INOUT part\_h is the partition object handle.

Tuples[] === left\_v, left\_x, right\_x, right\_v === \_v is overlap, \_x is index

DRI\_group\_create (n-processes, procs[], &group\_h)

// Create a process set (group)

IN n-processes is number of processes in the group,  
IN procs is the array of processes  
INOUT group\_h is a group object handle

DRI\_dist\_create (gdo\_h, part\_h, group\_h, &dist\_h)

// Create a distribution object

IN gdo\_h is the gdo object handle  
IN part\_h is the partition object handle  
INOUT dist\_h is the distribution object handle

This call is collective

Creates a distribution object from gdo, partition, and group

DRI\_layout\_create (n-dimensions, strides[], &layout\_h)

// Create a layout object

IN n-dimensions is dimensions  
IN strides is the array of strides for each dimension,  
INOUT layout\_h is the layout object handle

creates enumerated type of permutations of dimension combinations e.g., 3 d has 6 combinations .  
orientations and strides (strides, layout type, order (array of dimensions, order,

DRI\_xfer\_create (xfer\_name, dist\_h, layout\_h, n-buffs, buffs[], &xfer\_h)

// Create a transfer object

IN xfer\_name is a text symbolic name for the transfer

IN Dist\_h is the distribution object handle,

IN Layout\_h is the layout object handle

IN n-buffs is the number of buffers to allocate for the transfer

*INOUT* buffs[] is the array of buffers for the transfer

*INOUT* xfer\_h is the transfer object handle.

DRI\_do (xfer\_name, xfer\_h)

// the do call

IN xfer\_name is the name of a transfer to execute

IN xfer\_h is the transfer object handle.