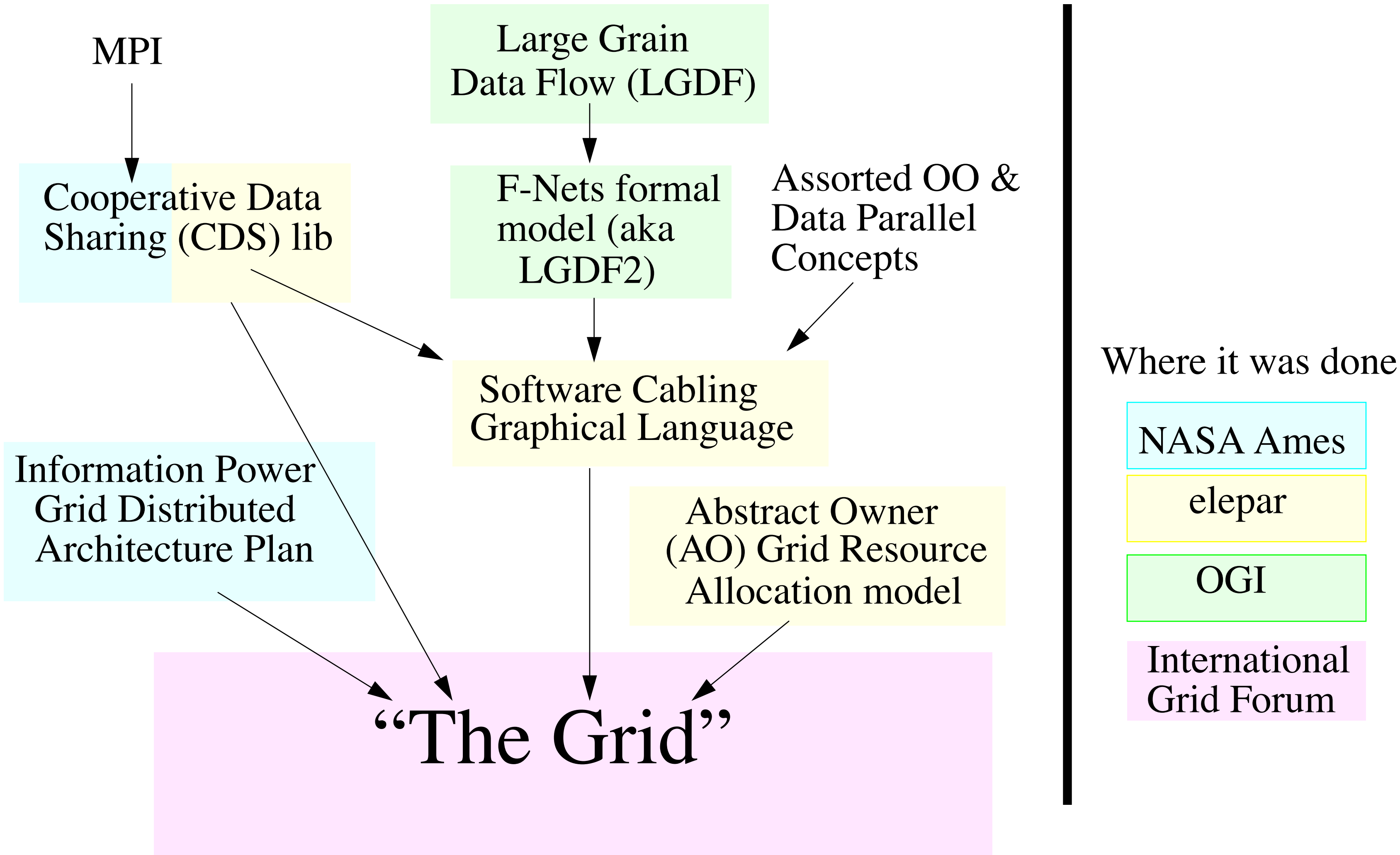
Cooperative Data Sharing (CDS): A Library Interface for Building Scalable Portable Parallel Programs

elepar
Beaverton, OR
moreinfo@elepar.com
http://www.elepar.com

How CDS Fits Into The Big Picture



Why Message Passing

Usually used for high-latency (e.g. distributed memory) architectures, because:

- **Messaging data to local memory (from distant) decreases costly remote accesses over slow interconnect**
- **Initiation of transfer before destination needs data decreases lag caused by interconnect latency**

Usually not used for low-latency (e.g. shared bus) architectures, because:

- **Copying data often serves no purpose, just increases latency, decreases bandwidth**
- **Initiating message before receiver offers a buffer requires additional buffering, which may serve no purpose.**

Why Shared Memory

Usually used for low-latency (e.g. shared-bus) architectures, because:

- **Easier to access each datum where it is**
- **Doesn't hurt much to experience latency of demand and delivery for each**

Usually not used for high-latency (e.g. distributed-memory architectures) because:

- **Cannot move data toward next processor before it is requested (to hide latency) even if previous process knows where it will be needed next**
- **Requests are always made in small granularity, so multiple requests must be made to move much data, each datum experiencing the latency of the interconnect twice**

Comparison of Semantic Options

Model--> / Semantic option available	MPI	Linda	RK	(D)SM	CDS
Non-destructive write (enqueue)	X	X	X		X
Destructive write (overwrite)				X	X
Destructive read (dequeue)	X	X	X		X
Non-destructive read (read)		X		X	X
Keep copy of communicated data	X	X			X
Dont “ “			X	X	X
Identify consumer	X	*	X		X
Dont “ “		X		X	X
Identify producer	X	*			X
Dont “ “	X	X	X	X	X

* Linda uses preprocessor to identify these automatically

Cooperative Data Sharing (CDS) Goals

Can't continue to program as though we know what the latency relationship will be between each process at runtime!

- **Architectures are getting more complex (e.g. clusters, grids), so even on a single architecture, there may be a mix of latencies**
- **Moving to a new architecture (even to a uniprocessor) means rewriting application**
- **Even when "native" semantics aren't optimal, they are still sometimes desirable**

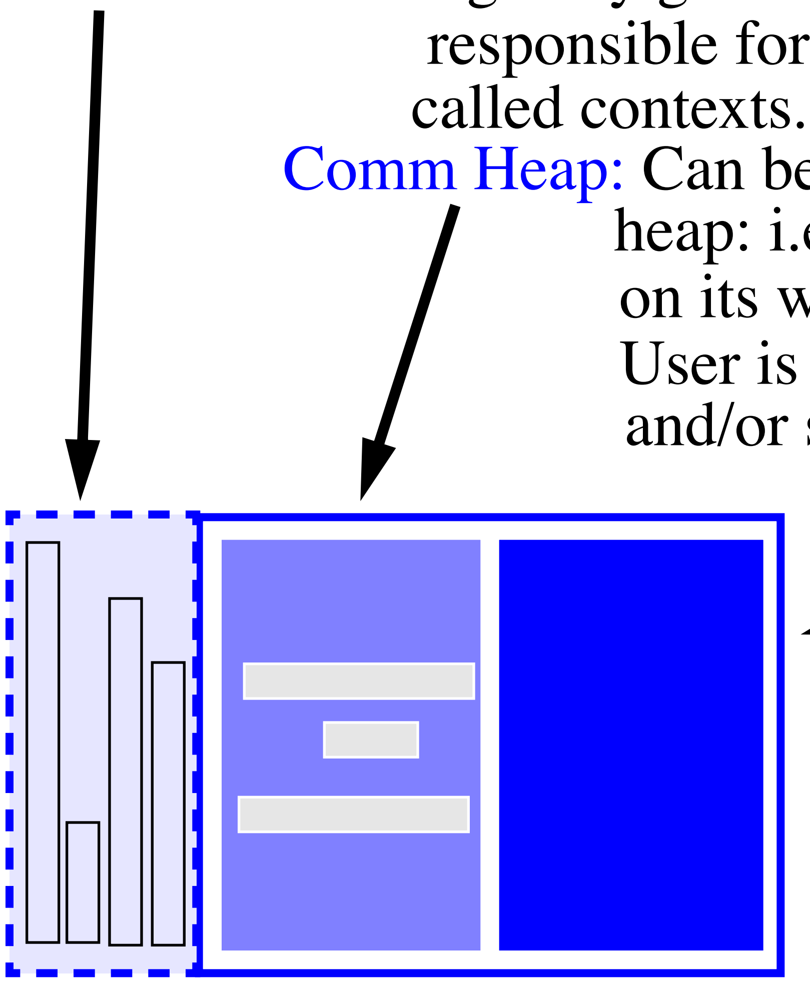
Main idea: Let programmer specify desired semantics for EACH communication (i.e. policy), let runtime system optimize it based on current architectural and run-time state (i.e. mechanism)

AND KEEP IT SMALL AND SIMPLE!

A Process in CDS (Logically)

Comm Cells: Logically global set of queues (of regions). User is responsible for creating and deleting, in groups called contexts.

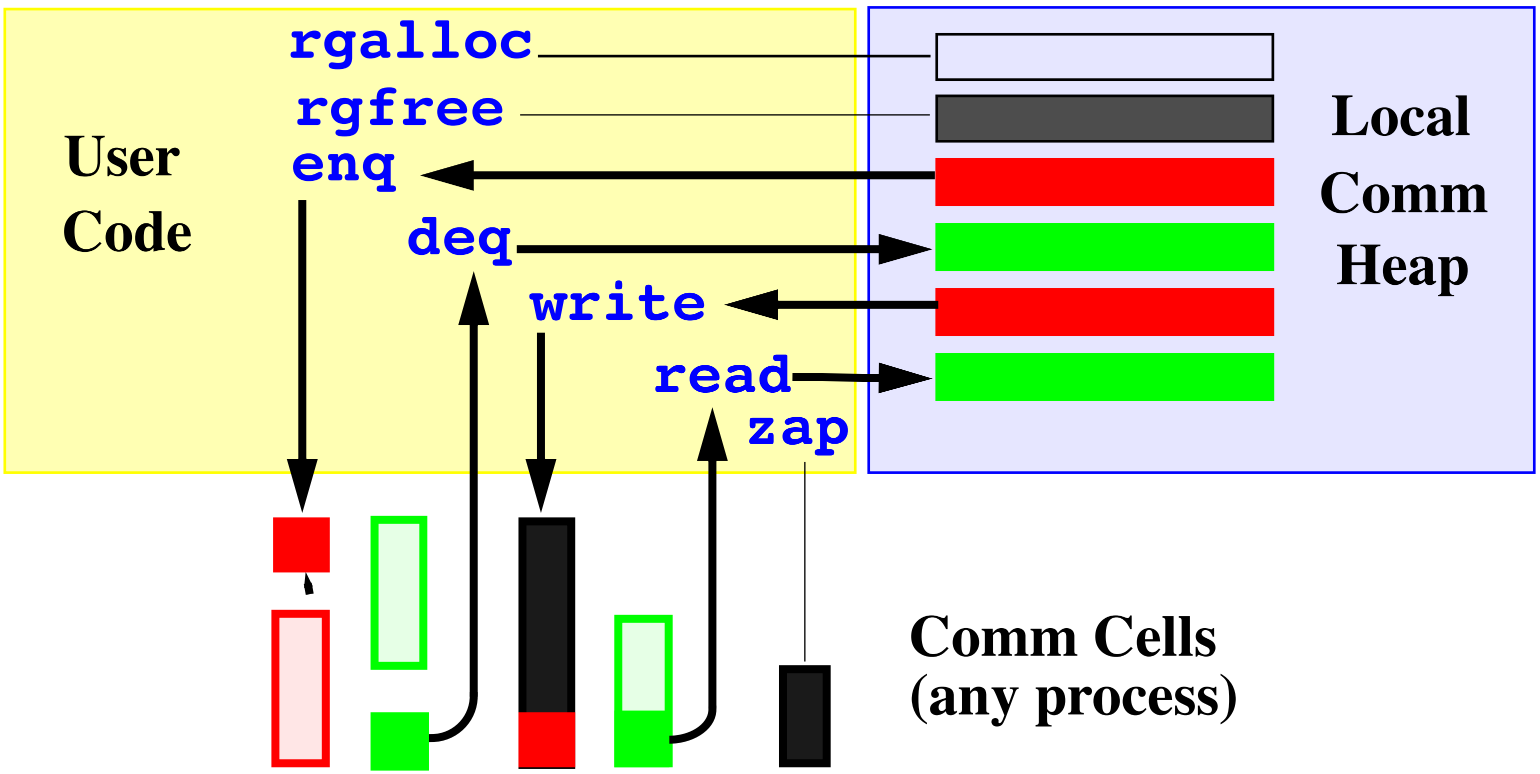
Comm Heap: Can be treated like standard heap: i.e. **`malloc`**, **`free`**. Holds data on its way to or from a Comm Cell. User is responsible for enlarging and/or shrinking.



User code & data: Standard Unix process.

Primitives logically move data between local comm heap and any comm cell.

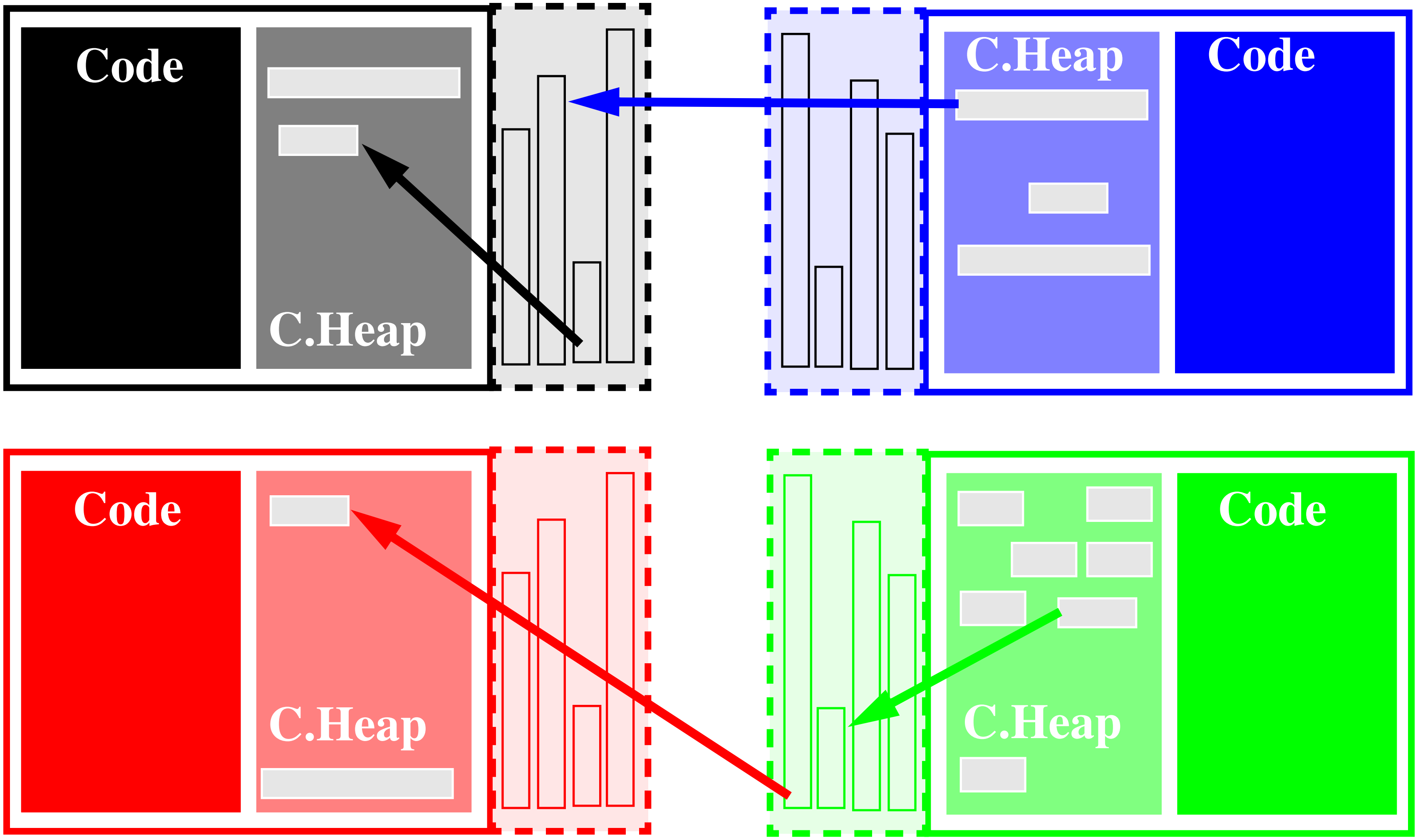
Basic CDS Primitives (Logically)



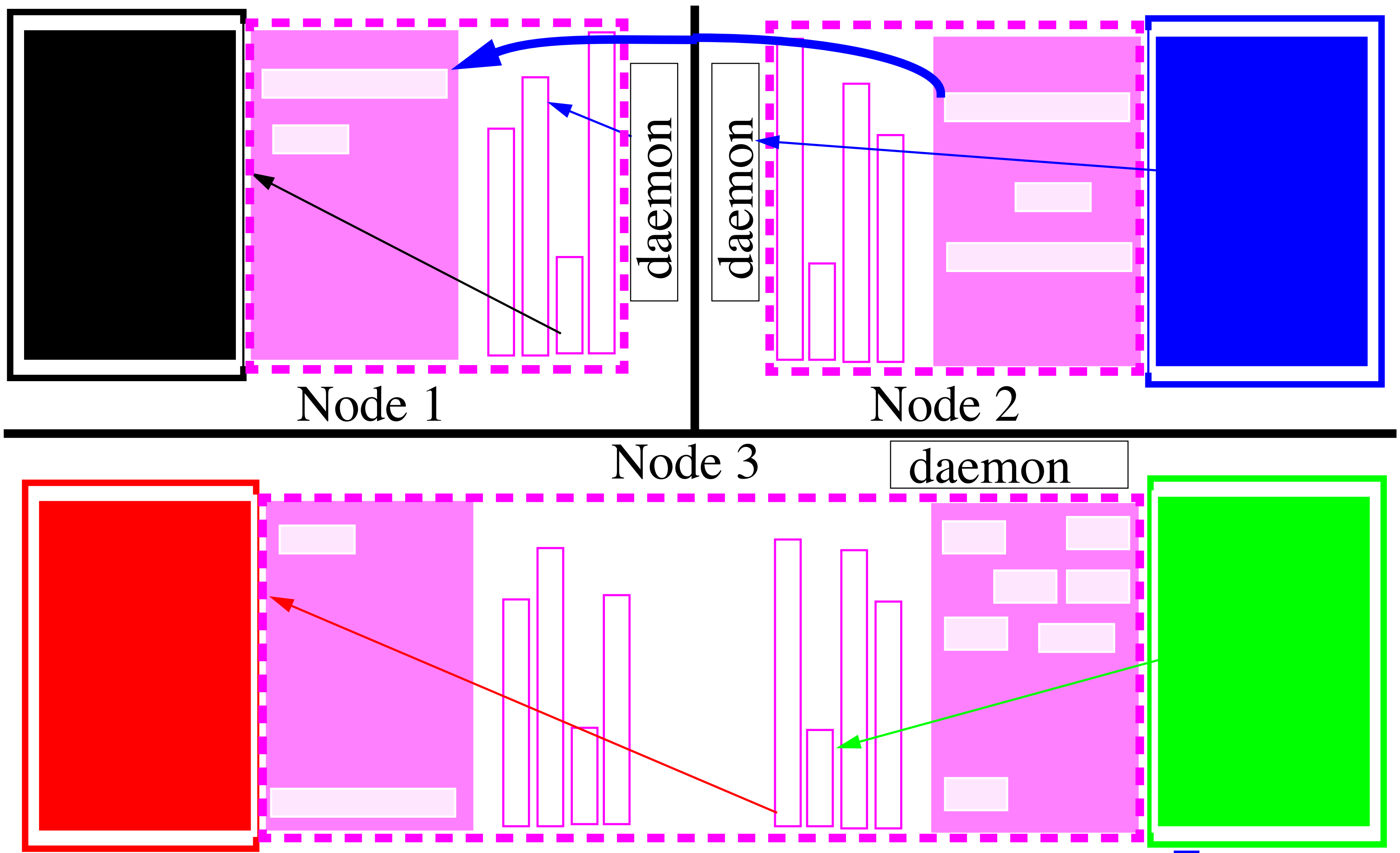
enq & write are cases of put; deq & read of get

Most have a timeout value, and puts can also perform rgfree

Basic CDS Primitives (Logically)



CDS Primitives (Physically)



Making Logical & Physical Conform

Same physical region may end up as multiple logical regions! So...

If a process is going to modify a region, it must notify CDS first!
(unless the region has never been shared)

Notify CDS using special arguments on standard CDS primitives, or a special `rgmod` primitive for other cases

How it works (fairly standard "Copy on write"):

- **Each physical region contains a hidden reference count**
- **If other refs to a region exist when modification is requested, a copy is made (under the covers)**
- **Region IDs are handles (i.e. pointers to pointers), so region ID itself doesn't change even if copy occurs**

A Few Extras

`benq` like `enq`, but blocks until

- the cell being written to is empty and
- there is a pending `deq` for the cell

...so, with a zero time-out, similar to MPI "ready send", with infinite time-out, similar to MPI "synchronous send"

`enqm`, `writem`, and `benqm` are multicast versions of `enq`, `write`, and `benq`

`ideq`, `iread`, and `ibenq` are non-blocking versions of `deq`, `read`, and `benq`

CDS Basic Primitives: Syntax

```
write(rgid,proc,cntxt,cell,perm)
enq (rgid,proc,cntxt,cell,perm,blktime)
benq(rgid,proc,cntxt,cell,perm,timeout)
read(&rgid,proc,cntxt,cell,perm,timeout)
deq (&rgid,proc,cntxt,cell,perm,timeout)
ideq(&ithrd,&rgid,proc,cntxt,cell,perm,timeout)
zap(proc,cntxt,cell)
iread(&ithrd,&rgid,proc,cntxt,cell,perm,timeout)
ibenq(&ithrd,rgid,proc,cntxt,cell,perm,timeout)
wait(ithrd)
waitm(nthrds,ithrds)
rgalloc(size)
rgmod(rgid,blktime)
rgfree(rgid)
```

Process Initiation: enlist

`enlist` brings a process into CDS program (creating it if necessary)

- **New process has context w/1 comm cell, small comm heap**
- **Region can be put into cell as part of `enlist` operation**
- **New process is told ID of its enlistor**
- **Enlistor does not block, and is not told ID of new process**

Typical enlistor protocol:

- **Enlists process, passing it any necessary info**
- **If error region received, or no word comes in reasonable time, something went wrong**

Typical new process protocol:

- **Augment comm heap and add contexts as necessary**
- **Report presence to another process (often parent) with put**

Handlers ("Interrupts")

Any cell can be augmented with a high or low water mark, and a handler (i.e. function). Handlers (and the "mainline" or main thread) can be given priorities.

Low-water mark: Handler initiated before `get` if cell contains that number of regions or fewer. (Consider 0.)

High-water mark: Handler executed after `put` if cell contains that number of regions or more. (Consider 1.)

Handler may be executed in a new thread, or within current thread (as coroutine). In latter case, thread initiations and switches occur only when CDS is entered for some reason.

Copying and Heterogeneous Communication

`copyto`, `copyfm`, `copytofm` perform

- copying between memory and CDS region (in comm heap) or 2 CDS regions
- packing and unpacking (marshalling/demarshalling)
- data translation (e.g. between heterogeneous representations)

For copying and packing, copy routines are controlled by nested sequence of integer triples, which serves same basic purpose as MPI type but easier to build and manipulate.

(offset,type,replicator)

`transtab` takes two process IDs and returns name of conversion table (suitable for use by copy routines) for translating between their data representations.

Shared Memory Semantics

`enq`ing a region into a cell makes that region accessible to other processes—i.e. like releasing a lock on the region

`deq`ing a region from a cell makes that region accessible to you, but removes accessibility to other processes—like acquiring a lock

Additional (macro-like) extensions simplify using CDS in a "shared memory with release consistency style":

Function/Macro	Meaning	Translates Into
`acqwl`	Acquire write lock	deq w/modify
`rlswl`	Release write lock	write, rgfree
`acqrl`	Acquire read lock	read (no modify)
`rlsrl`	Release read lock	rgfree
`wl2rl`	Convert write lock to read lock	write (no modify)

Also asynchronous routines `iacqwl`, `iacqrl`

Message Passing Semantics

Consider additional composite functions defined as follows:

Function	Meaning	Semantically identical* to
`send`	Send message	`rgalloc, copyto, enq, rgfree`
`recv`	Receive message	`deq, copyfm, rgfree`
`sendx`	Destructive send	`rgalloc, copyto, write, rgfree`
`recvx`	Non-destructive recv	`read, copyfm, rgfree`

* except lack of comm heap space may not result in error

- **Region used does not exist before or after function call**
  --> the region can be optimized out in some cases
  --> these can be optimized in all the same ways as MPI
- **Real reason that `copy` routines need to be in CDS**
- **Should be used whenever they match users needs exactly**

The CDS Interface (49 for now)

Managing comm heap and contexts/cells

`rgalloc rgmod rgfree rgsize rgrealloc`

`addcntxt delcntxt grwcntxt`

Communication primitives

`read deq benq enq write zap enqm writem`

`iread ideq ibenq wait waitm ienqm benqm`

Copying and Translation

`copyto copyfm copytofm transtab`

Composite functions (shared mem and msg passing)

`recv bsend recvx send sendx sendm sendxm`

`acqrl acqwl rlsrl rlswl wl2rl`

`irecv ibsend irecvx iacqrl iacqwl`

Process and thread control

`enlist init myinfo hdlr prior`

What CDS Isn't Special At

(Well, no better than shared memory or message passing)

- **Data parallelism**
- **Variable Granularity**
- **Object-oriented Programming**
- **Program Proving**
- **Fault Tolerance**
- **Software Engineering (Programming "in the large")**
- **Managing non-determinism**

F-Nets and Software Cabling

F-Nets is a formal model for parallel computers

- **Similar to Petri Nets, Data Flow, Turing machines, CA**
- **Good for reasoning about parallel programs & taming nondeterminism**
- **Can be considered as CDS-like communication style between atomic transactions (e.g fault tolerant)**

Software Cabling is graphical coordination language, based on F-nets

- **Object-oriented**
- **Data parallel**
- **Individual modules can be written in nearly any language**

--> Software Cabling solves the big problems that CDS alone doesn't

CDS Status and Plans

Status: Prototype exists (one at NASA?, one at elepar)

- **Uses UDP/IP and various shared memory**
- **Implemented on MP SGI & Sun workstations, Linux PC**
- **Speed can still be improved, through optimizations such as lock-free queues and spin-free locks**
- **Most of the work has been in interface definition**

Plans: elepar wants to be a user! What is best way?

- **Invest time and money (for every platform), and try to sell?**
- **Standardize, and let vendors implement themselves? (Probably by customizing existing MPI or other native communication packages) PICIS BOF at SC97**
- **Purely open source?**