A uniform programming model for complex distributed data objects in distributed and shared memory

V. Balaji
Princeton University and NOAA/GFDL
Robert W. Numrich
Minnesota Supercomputing Institute

ECMWF High Performance Computing Workshop
Reading, UK
29 October 2004

Overview

- A uniform view of existing memory models.
- Target codes: shallow water example.
- High-level syntax for expressing data dependencies.
- Implementation in various parallelism idioms.
- Advanced features.

Memory models

Shared memory signal parallel and critical regions, private and shared variables. Canonical architecture: UMA, limited scalability.

Distributed memory domain decomposition, local caches of remote data ("halos"), copy data to/from remote memory ("message passing"). Canonical architecture: NUMA, scalable at cost of code complexity.

Distributed shared memory or ccNUMA message-passing, shared memory or remote memory access (RMA) semantics. Processor-to-memory distance varies across address space, must be taken into account in coding for performance. Canonical architecture: cluster of SMPs. Scalable at large cost in code complexity.

Target codes

What algorithm developers need is to be able to think about the problem in terms of their needs, and not those of the underlying architecture. The language for expressing concurrent computation should not involve “messages”, “critical regions”, “processor-memory distance”, but instead, “data dependency”, i.e “in order to compute `a(i,j)` I need the value of `a(i,j+1)`”.

The target codes being considered are those with well-defined and predictable data dependencies. We have called these “grid codes”, the grid being the index-space substrate on which data dependencies are defined.

A general communication and synchronization model for parallel systems

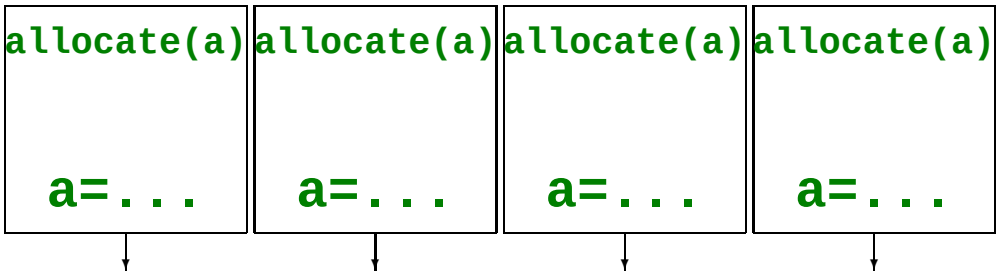
We use the simplest possible computation to illustrate the uniform memory model. Consider the following example:

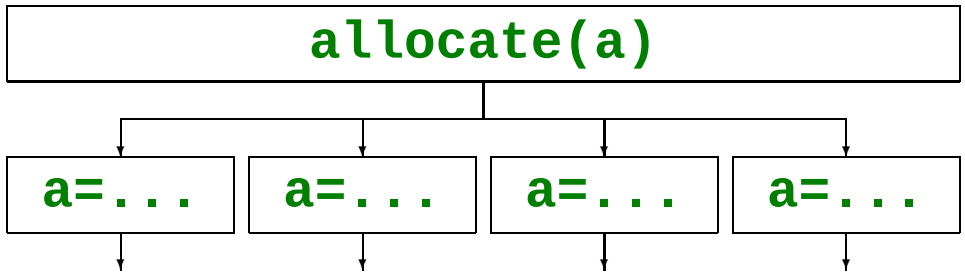
```
real ::  a, b=0, c=0
b = 1
c = 2
a = b + c
b = 3
```

(1)

at the end of which both **a** and **b** must have the value 3.

A key abstraction: PETs and TETs



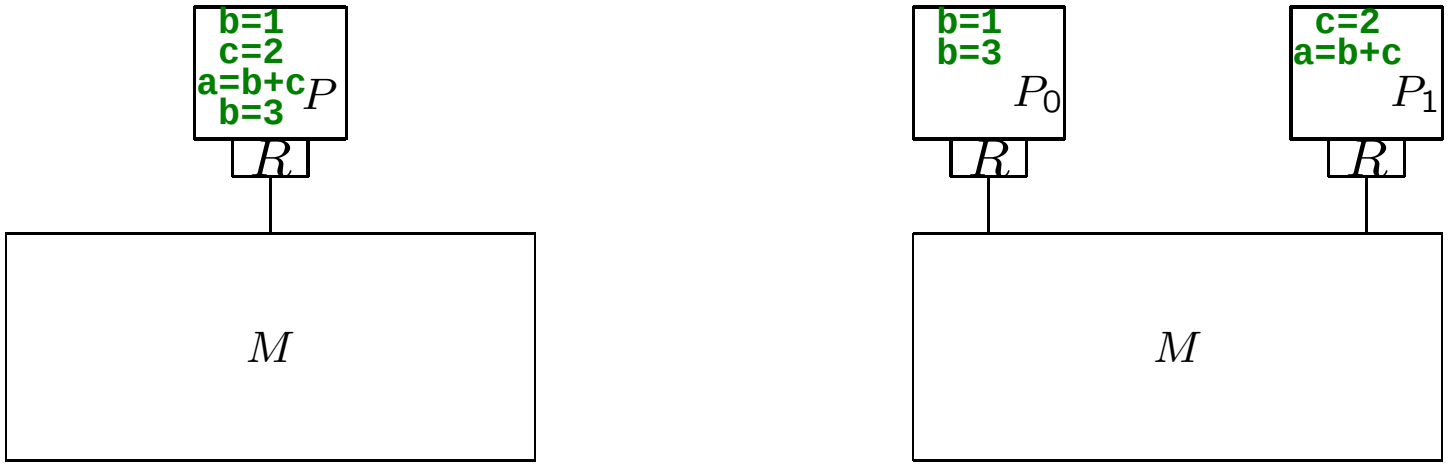


A ***persistent execution thread (PET)*** executes an instruction sequence on a subset of data in unison with other PETs.

The ***persistence*** requirement is that the thread must have a lifetime at least as long as the distributed data object.

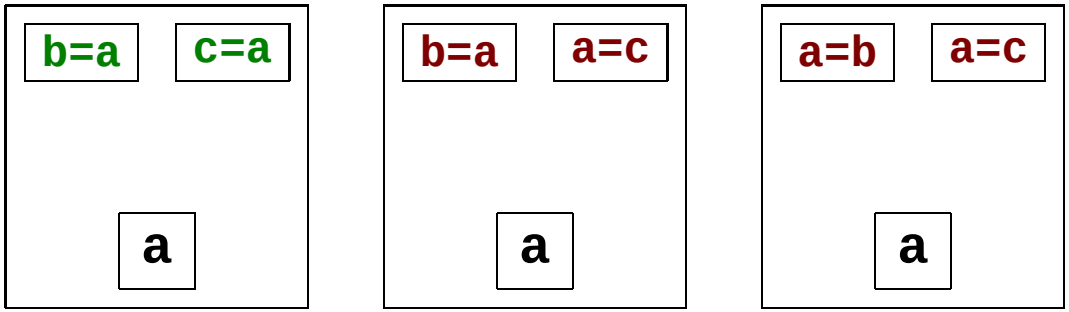
Persistent and transient execution threads. PETs exist prior to the allocation of any data object they operate upon; TETs are created after. PETs and TETs may be layered upon each other (not shown).

Sequential and parallel processing



Let us now suppose that the computations of **b** and **c** are expensive, and have no mutual dependencies, and are thus good targets for being performed concurrently. If there were two processors able to access the same memory, we could process **b** and **c** independently, as shown on the right. Two issues arise. One minor one is that the memory traffi c is somewhat increased: on the left the values of **b** and **c** can stay in the registers, without updating the memory values. On the right it is necessary, fi rst, to transfer the value of **b** to memory where it can be read by the other processor, and second. to signal that the computations is done. If the processor on the right reads the value of **b** before it has been updated (a ***race condition***) the result will be incorrect.

Race conditions



Race conditions occur when one of two concurrent execution streams attempts to write to a memory location when another one is accessing it with either a read or a write: it is not an error for two PETs to read the same memory location simultaneously. The second and third case result in a race condition and unpredictable results. The third case may be OK for certain reduction or search operations, defi ned within a **critical region.**

The central issue in parallel processing is the ***avoidance of such a race condition with the least amount of time spent waiting for a signal***: when two concurrent execution streams have a mutual dependency (the value of **b**), how does one stream know when a value it is using is in fact the one it needs? Several approaches have been taken.

Shared memory and message passing

Parallel processing: comparison of signals for shared-memory and message-passing. The computations **b=1** and **c=2** are concurrent, and their order in time cannot be predicted. (a) In shared-memory processing, mutex locks are used to ensure that **b=1** is complete before P_1 computes **a=b+c**, and that this step is complete before P_0 further updates **b**. In message-passing, each PE retains an independent copy of **b**, which is exchanged in paired send/receive calls. After the transmission, P_0 is free to update **b**.

Remote memory access (RMA)

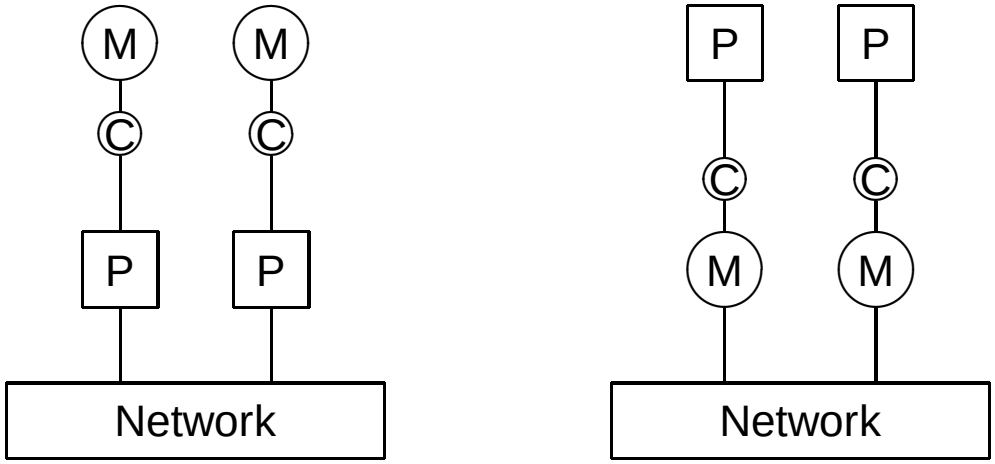
The name ***one-sided message passing*** is often applied to RMA but this is a misleading term. Instead of paired send/receive calls, we now have transmission events on one side (**`put`**, **`get`**) paired with ***exposure*** events (**`start`**,**`wait`**) and (**`post`**,**`complete`**), respectively, in MPI-2 terminology, on the other side. It is thus still "two-sided". A variable exposed for a remote **`get`** may not be written to by the PE that owns it; and a variable exposed for a remote **`put`** may not be read.

Note that P_1 begins its exposure to receive **`b`** even before executing **`c=2`**. This is a key optimization in parallel processing, ***overlapping computation with communication***.

Shared memory and message passing

Parallel processing: the full synchronization model applied to shared-memory protocols and message-passing. Note that in shared memory, P_0 never relinquishes the write-lock. In message-passing, this is done using non-blocking **`isend`** and **`irecv`** operations.

Non-blocking communication



- On ***tightly-coupled systems***, independent network controllers can control data flow between disjoint memories, without involving the processors on which computation takes place. True non-blocking communication is possible on such systems.

- Note that caches induce complications.

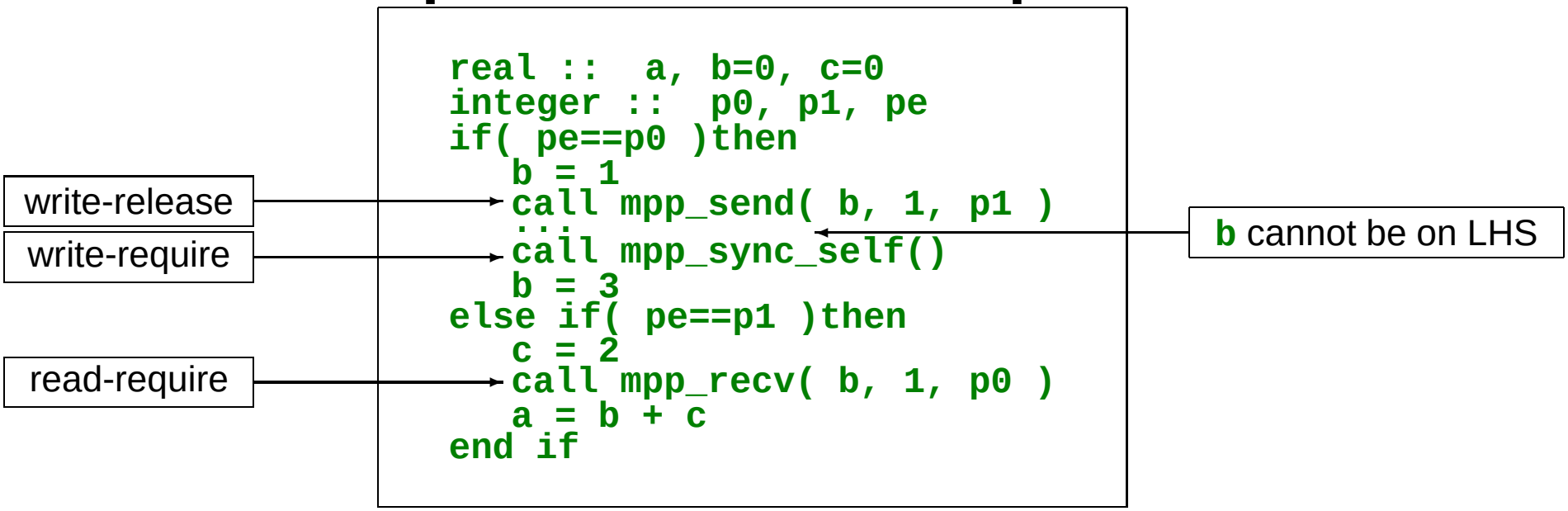
- On ***loosely-coupled systems***, this is implemented as the semantically equivalent ***deferred communication***, where a communication event is registered and queued, but only executed when the matching block is issued.

The complete memory model

All the above mechanisms of sharing data can be combined into a single uniform memory model, where an object has two states **`READ`** and **`WRITE`**. There are three types of access operations, **`request`**, **`require`**, and **`release`**.

Access	State	MPI	shmem	MPI-2	Threads
Request	READ	irecv	status=WAIT	post	WRLOCK?
Require	READ	wait	wait(status=OK) unbuffer put(put=OK)	wait	wait(!WRLOCK) lock RDLOCK
Release	READ				unlock RDLOCK
Request	WRITE	isend	wait(put=OK) put(buffer) put=WAIT	start put	RDLOCK?
Require	WRITE	wait	fence put(status=OK)	complete	wait(!RDLOCK) lock WRLOCK
Release	WRITE				unlock WRLOCK

Example: a release-write protocol



Any existing memory model can be formulated as a subset of the full UMM described above. For example, the FMS MPP layer is based on a non-blocking **`mpp_send`** and a blocking **`mpp_recv`** call.

That's all very well, but...

The full UMM as applied to a single scalar variable **b** appears rather elaborate. In real codes, however, we are dealing with more complex data objects, e.g distributed arrays, or components. The strength of this model becomes more apparent if we build UMM semantics directly into high-level distributed datatypes.

The existing and emerging modeling frameworks already implement such distributed data objects.

Distributed array An **ESMF_Field** holds its distributed contents in an **ESMF_Array**, which already contains metadata describing distribution information (**DataMap**, **Layout**, **VM**). The actual (***"naked"***) local array can be attached and detached with no data copies.

```
type(ESMF_Field) ::  field
real ::  a(:,:,:)
a(:,:,:)  => ESMF_FieldGetDataPointer(field)
call ESMF_FieldHalo(field)
...  = a(i+1,j+1,k) + ...
```

(2)

Components Components and their states are distributed entities whose parallel data exchange model is a subset of the full UMM. For instance, PRISM exchanges data using a non-blocking **PRISM_Put** and a blocking **PRISM_Get** (a ***"write-release"*** model).

Programming the UMM

- Clearly we would like to keep the complexities of programming distributed data objects out of numerical algorithms as much as possible. We are currently exploring a kernel-driver programming model, where each algorithm is coded as a module with a driver and a set of kernels. External routines call the driver, using high-level distributed objects; it in turn orchestrates numerical kernels that are written with simple intrinsic data types passed by reference.

```
type(ESMF_Field) ::  field
real ::  a(:,:,:)
a(:,:,:)  => ESMF_FieldGetDataPointer(field)
call ESMF_FieldHalo(field)
...  = a(i+1,j+1,k) + ...
```

(3)

- The process of acquiring access to an array's contents has been split into a ***request*** and ***require*** phase in order to enable useful work on a PET while waiting for access. This may be done by a compiler ("pre-fetching"). Programming language extensions such as Co-Array Fortran are designed for declaration of any derived type as a distributed object, and allowing the compiler to apply pre-fetching and other optimizations.

Example: 1D shallow water model

$$\begin{aligned} \frac{\partial h}{\partial t} &= -H\frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial t} &= -g\frac{\partial h}{\partial x} \end{aligned} \tag{4}$$

A forward-backward shallow water code might look like:

$$\begin{aligned} h_i^{t+1} &= h(h_i^t, u_i^t, u_{i-1}^t, u_{i+1}^t) \\ u_i^{t+1} &= u(u_i^t, h_i^{t+1}, h_{i-1}^{t+1}, h_{i+1}^{t+1}) \end{aligned} \tag{5}$$

```
BEGIN TIME LOOP:
   h(i) = h(i) - (0.5*H*dt/dx)*( u(i+1) - u(i-1) ) FORALL i
   u(i) = u(i) - (0.5*g*dt/dx)*( h(i+1) - h(i-1) ) FORALL i
END TIME LOOP:
```

(6)

Memory allocation

The first step in a discrete computation of Eq. 5 on a distributed set of PETs is to allocate memory for a distributed array:

```
MakeDistributedArray( NX, NP, (+1,-1), h, s, e )
```
(7)

- Discretize on **NX** points, distributed across **NP** PETs.
- Data dependencies are **(+1,-1)** (this data structure can be made arbitrarily complex).
- Each PET receives a pointer **h** to the local portion of the distributed array, as well as indices **s** and **e** marking the start and end of the local computational domain.

Implementation

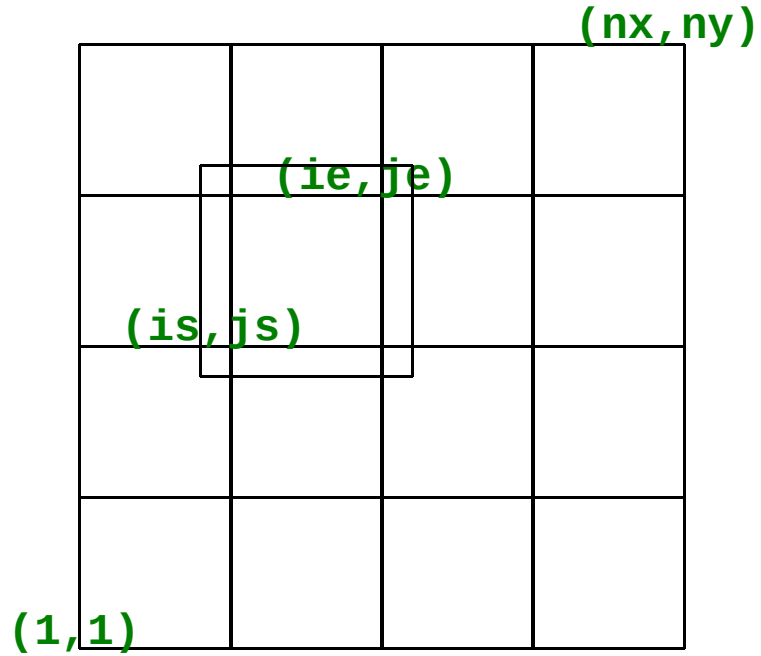
Distributed memory PET gets an array **`h(s-1,e+1)`**, which includes a local cache of the data on neighbouring PETs (the ***"halo"***). User must perform operations (***"halo updates"***) at appropriate times to ensure that the cache contains a correct copy of the remote data.

Shared memory one PET requests memory (**`malloc`**) for all **`NX`** points while the others wait. All PETs are given a view into this single array.

DSM or ccNUMA processor-to-memory distance varies. We call the set of PEs sharing flat (uniform) access to a block of physical memory an **mNode**, and the set of PEs sharing an address space an **aNode**.

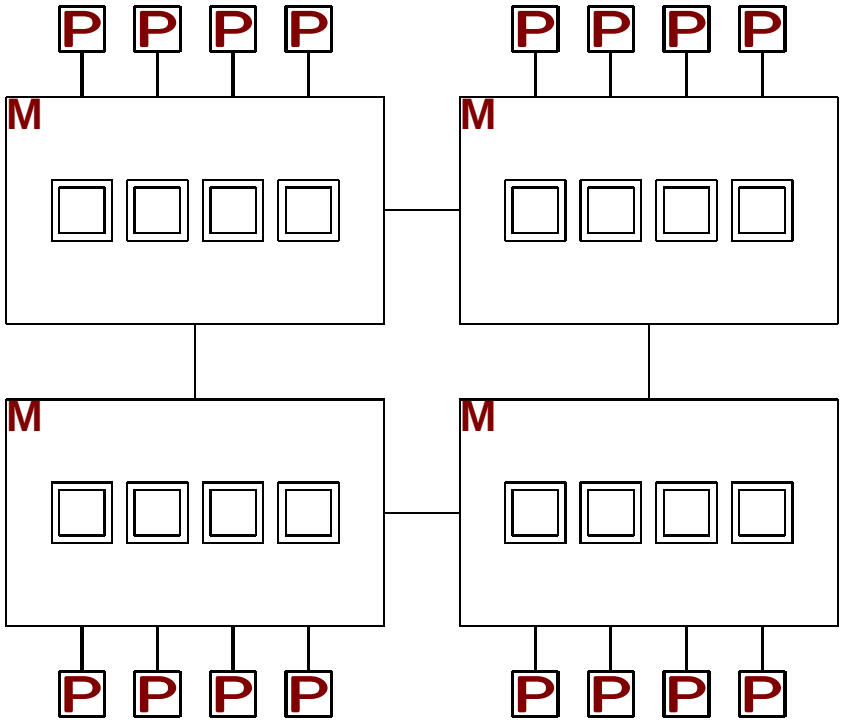
Whether it is optimal to treat an entire aNode as shared memory, or distributed memory, or some tunable combination thereof depends on the platform (hardware), effi ciency of the underlying parallelism semantics for shared and distributed memory (software), and even problem size. Memory request procedures must be flexible.

A 2D example



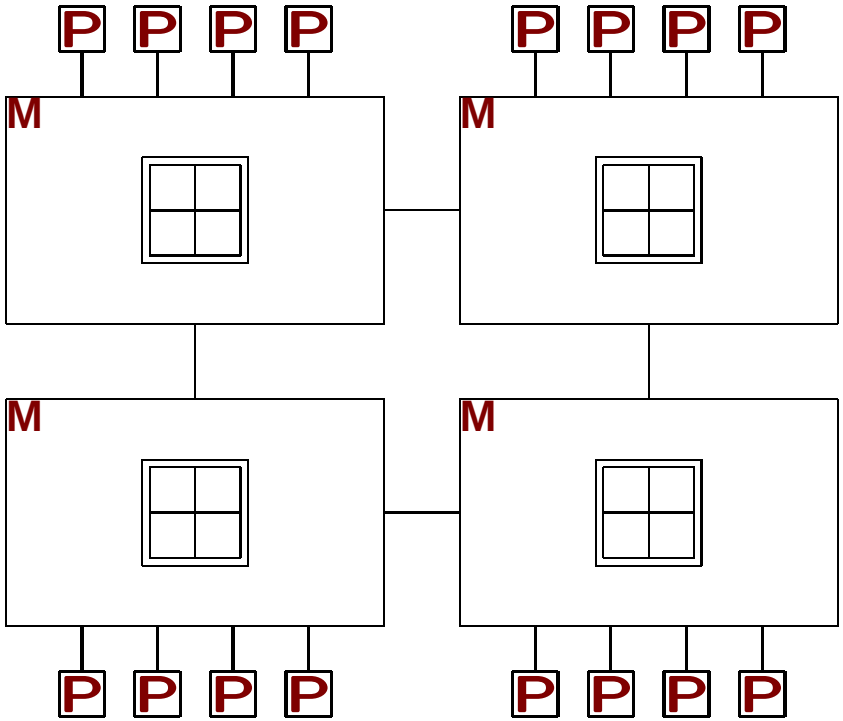
Consider a platform consisting of 16 PEs consisting of 4 mNodes of 4 PEs each. We also assume that the the entire 16-PE platform is a DSM or ccNUMA aNode. We can then illustrate 3 ways to implement a **`DistributedArray`**. One PET is scheduled on each PE.

Distributed memory



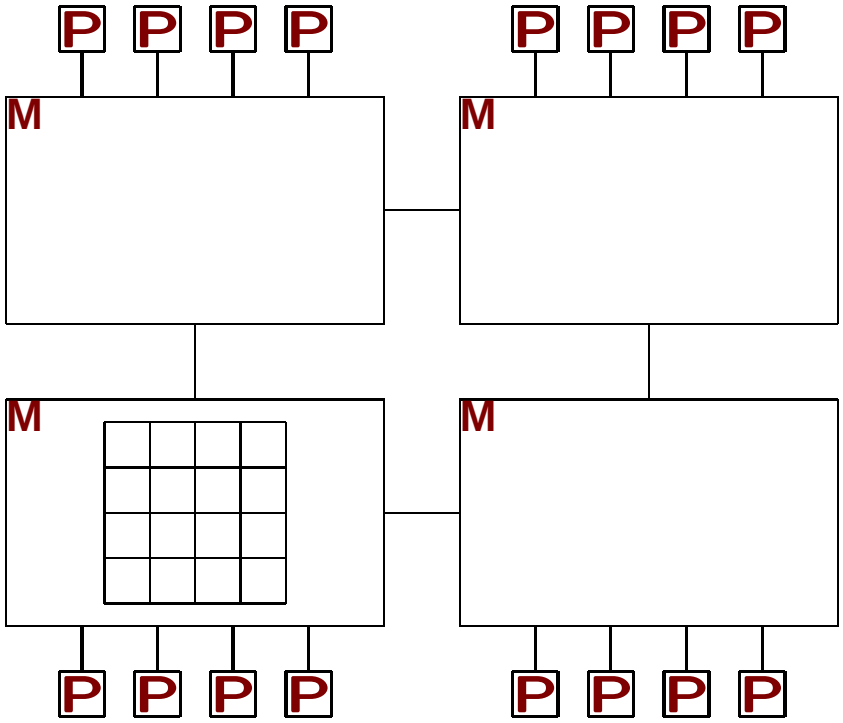
- each domain allocated as a separate array with halo, even within the same mNode.
- Performance issues: the message-passing call stack underlying MPI or another library may actually serialize when applied within an mNode.

Hybrid memory model



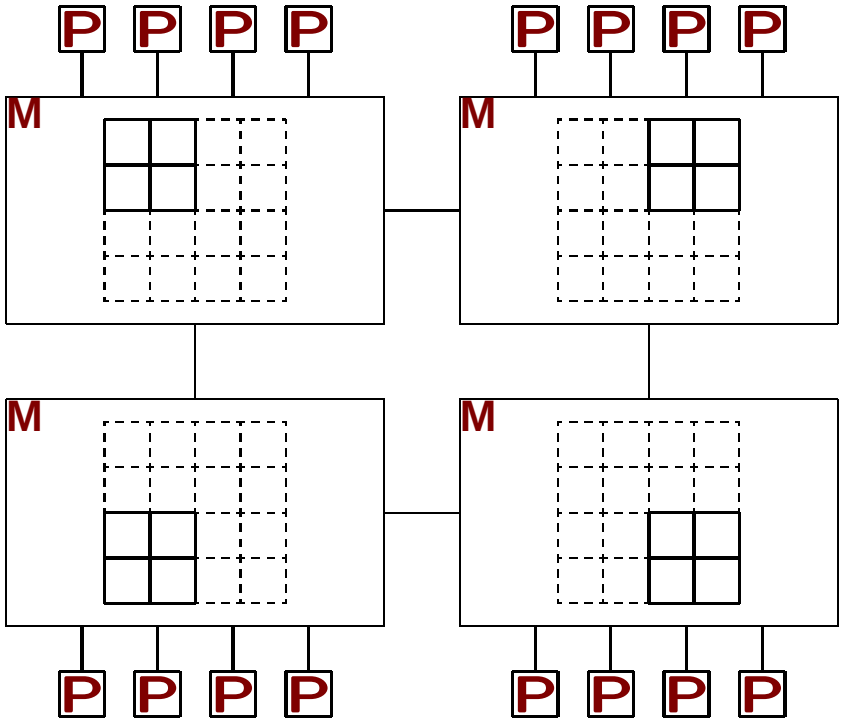
- shared across an mNode, distributed among mNodes.
- fewer and larger messages than distributed memory, may be less latency-bound.

Pure shared memory



Array is local to one mNode: other mNodes requires remote loads and stores. OK on platforms that are well-balanced in bandwidth and latency for local and remote accesses. ccNUMA ensures cache coherence across the aNode.

Intelligent memory allocation on DSM



Better memory locality: allocate each block of 4 domains on a separate page, and assign pages to different mNodes, based on processor-memory affi nity.

Memory access operations for distributed arrays

Request WRITE must be posted (and fulfilled: see **Require** below) before ***array*** appears on LHS. Non-blocking.

Request READ must be posted (and fulfilled: see **Require** below) before ***array halo*** appears on RHS. Non-blocking.

Require WRITE must be fulfilled before ***array*** appears on LHS. Blocking.

Require READ must be fulfilled before ***array halo*** appears on RHS. Blocking.

Release WRITE must be posted immediately following update of ***array*** contents. Non-blocking.

Release READ must be posted immediately following computations requiring ***array halo***. Non-blocking.

Vectorization!

Simultaneous **READ** and **WRITE** access to an array is never permitted: if a PET P has **WRITE** access to its computational domain, the neighbouring PET will have **WRITE** access to P's halo, and thus P cannot have **READ** access.

To put it another way, one cannot write a loop of the following form:

```
h(i) = a*( h(i+1) - h(i-1) ) FORALL i
```
(8)

as the result is not well-defined on distributed arrays. ***This is the same rule that must be obeyed to avoid vector dependencies***.

Summary

- Uniform syntactic view of distributed, shared and hybrid memory architectures for codes with well-defi ned and predictable data dependencies. No software-layering issues in hybrid programming.
- Underlying architecture is modeled as a set of ***persistent*** execution threads (PETs), having a lifetime at least as long as the distributed data object.
- Applicable to distributed arrays, as well as states and components.
- Access operations: `request`/`require`/`release`. `require` alone is blocking.
- Coding rules: same restrictions as for vectorization.
- Implementation: MPI, shmem, MPI-2, shared memory threads using mutex locks.
- Distributed data objects are well-suited to the use of the co-array extensions.
- A kernel-driver programming model is used to keep the extended datatypes restricted to the driver, and have kernels that use simple types passed by reference.