PSETs: a new approach to shared memory

V. Balaji
NOAA/GFDL and Princeton University

Princeton NJ
8 March 2006

A general communication and synchronization model for parallel systems

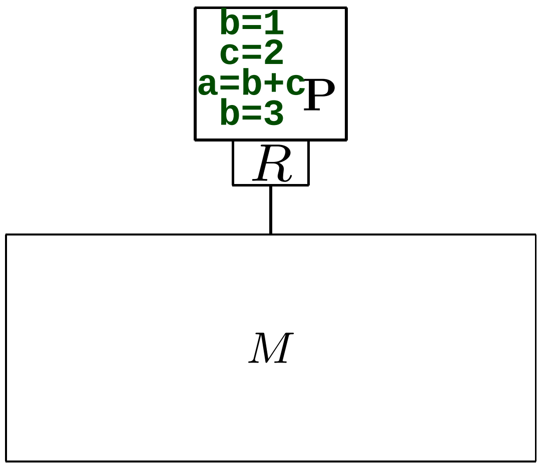
We use the simplest possible computation to compare shared and distributed memory models. Consider the following example:

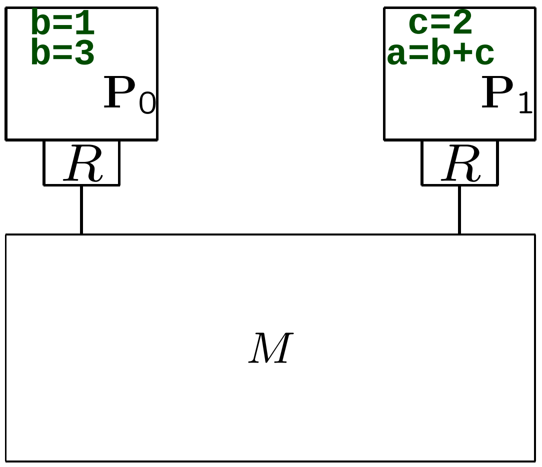
```
real ::  a, b=0, c=0
b = 1
c = 2
a = b + c
b = 3
```

(1)

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

Sequential and parallel processing



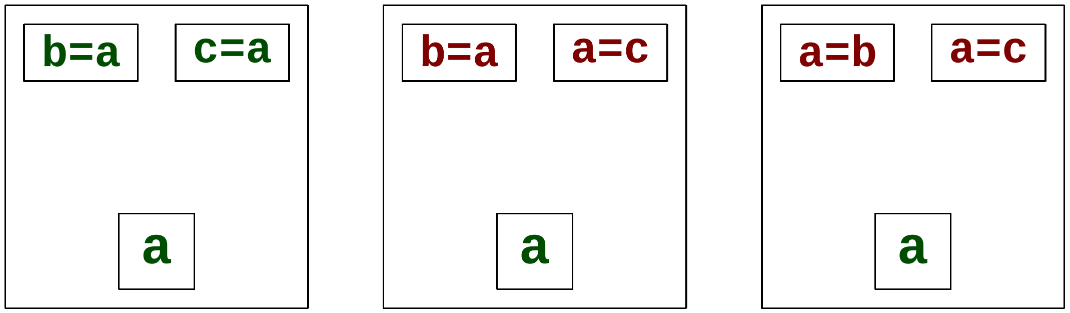


Let us now suppose that the computations of **b** and **c** are expensive, and have no mutual dependencies.

Then we can perform the operations ***concurrently***:

- Two PEs able to access the same memory can compute **b** and **c** independently, as shown on the right.
- Memory traffic is increased: to transfer **b** via memory, and to control the contents of cache.
- Signals needed when **b=1** is complete, and when **a=b+c** is complete: otherwise we have a ***race condition.***

Race conditions

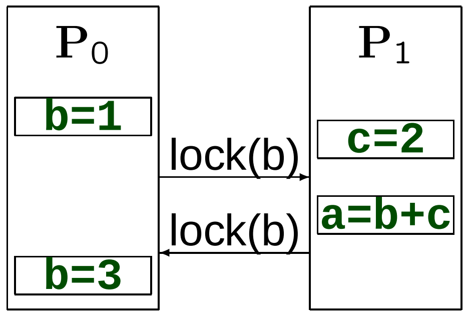


Race conditions occur when one of two concurrent execution streams attempts to write to a memory location when another one is accessing it with either a read or a write: it is not an error for two PEs to read the same memory location simultaneously. The second and third case result in a race condition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

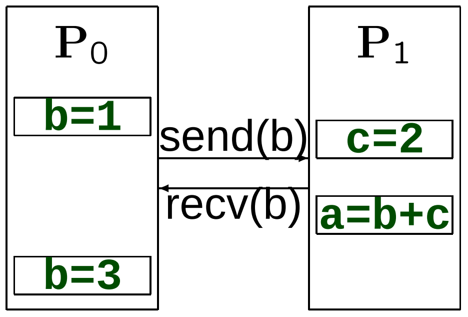
The central issue in parallel processing is the ***avoidance of such a race condition with the least amount of time spent waiting for a signal***: when two concurrent execution streams have a mutual dependency (the value of **a**), how does one stream know when a value it is using is in fact the one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

Shared memory and message passing

The computations **b=1** and **c=2** are concurrent, and their order in time cannot be predicted.



(a)

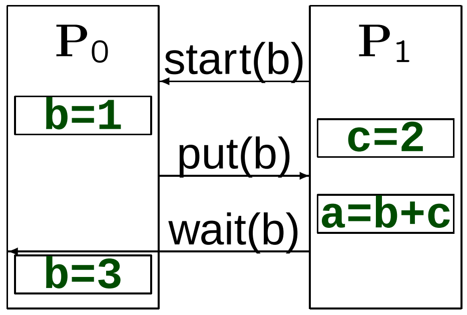


(b)

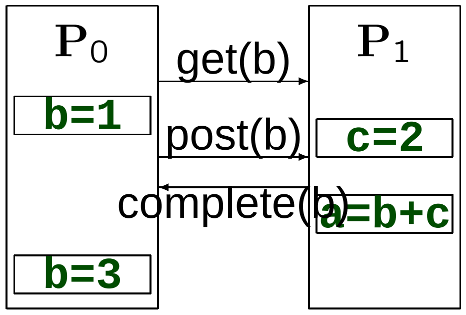
- In shared-memory processing, ***mutex locks*** are used to ensure that **b=1** is complete before P_1 computes **a=b+c**, and that this step is complete before P_0 further updates **b**.

- 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)



(a)

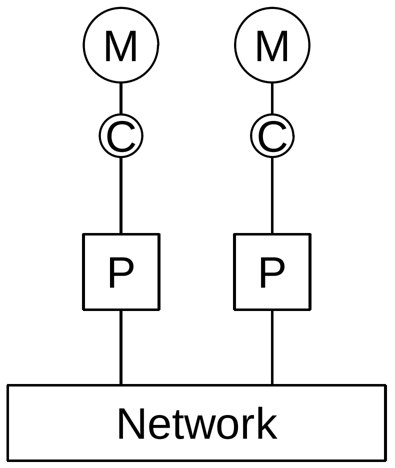


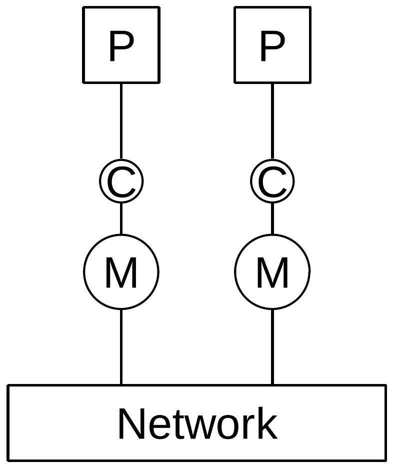
(b)

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***.

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.

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.

Platform survey

Including defunct ones, and not fully operational ones...

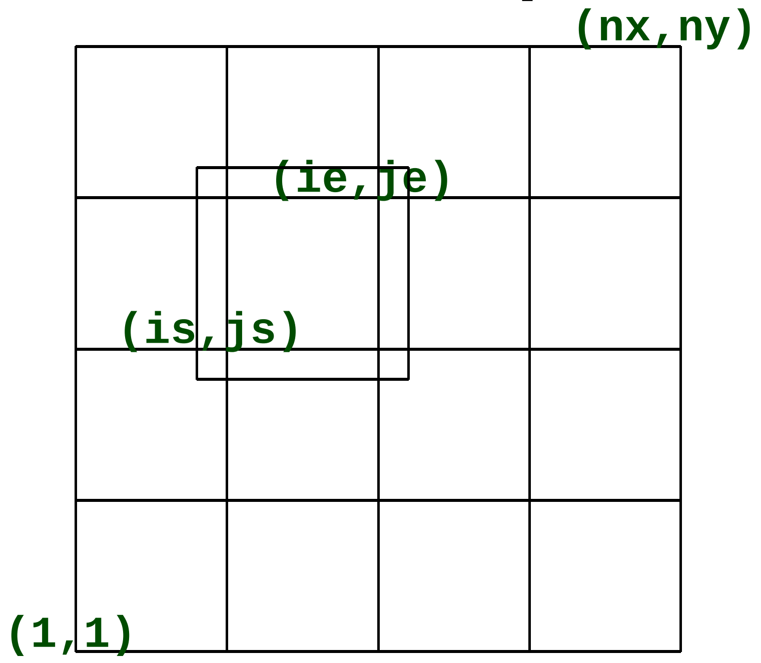
Shared memory Cray X-MP, Y-MP, C90, T90 (2-32p).

Distributed shared memory, ccNUMA SGI Origin, Altix (up to 2048p).

Distributed memory Cray T3E, XT3, Beowulf, IBM BlueGene.

Hybrid Cray X1 (32p), IBM SP (64p), SMP Beowulf ($\sim$4p).

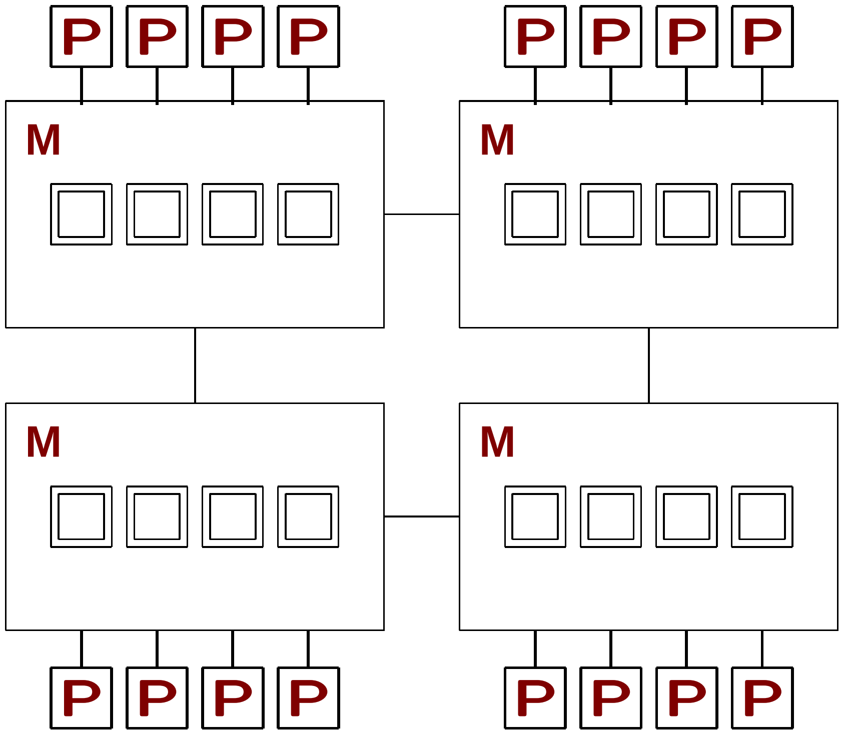
A 2D example



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 process is scheduled on each PE.

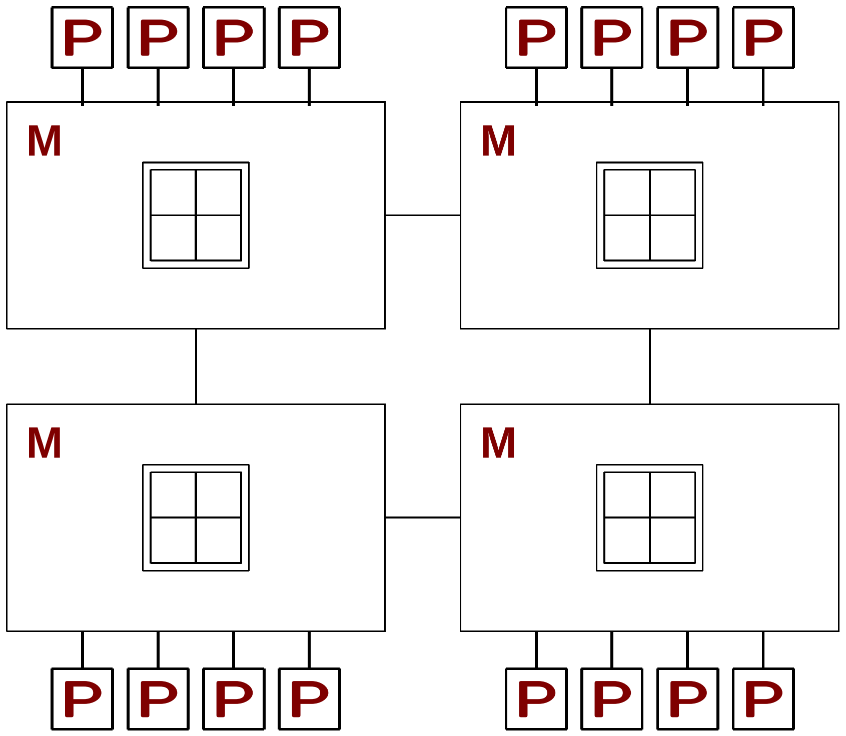
Hardware note: the SGI Numalink fabric maintains parity between local and remote bandwidth (∼2:1). On clusters 50:1 is more typical.

Distributed memory



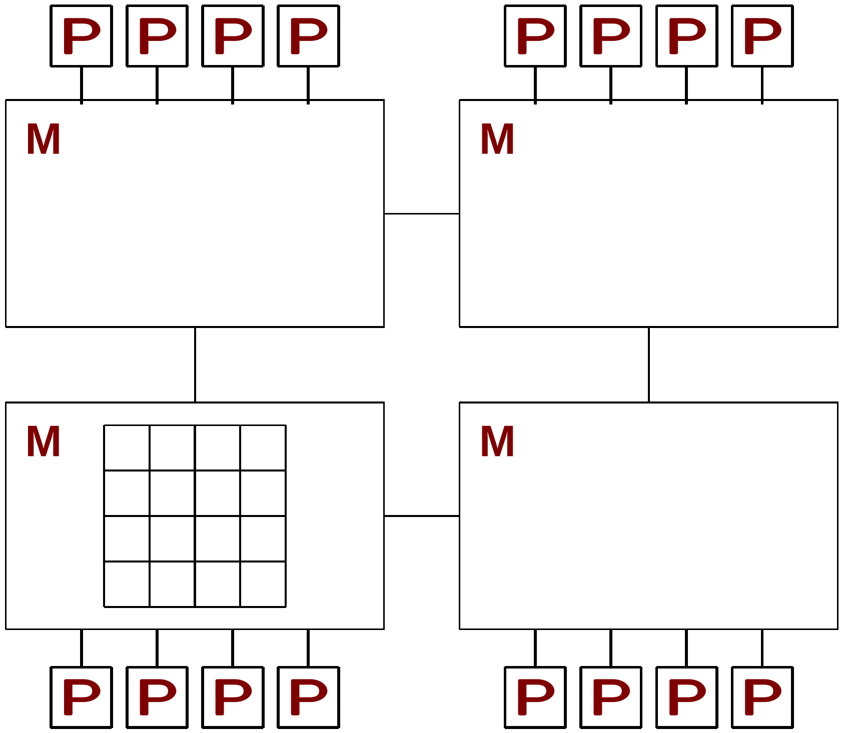
- each domain allocated as a separate array with halo, even within the same mNode.
- Performance issues: the message-passing call stack underlying MPI or another library may actually serialize when applied within an mNode.

Hybrid memory model



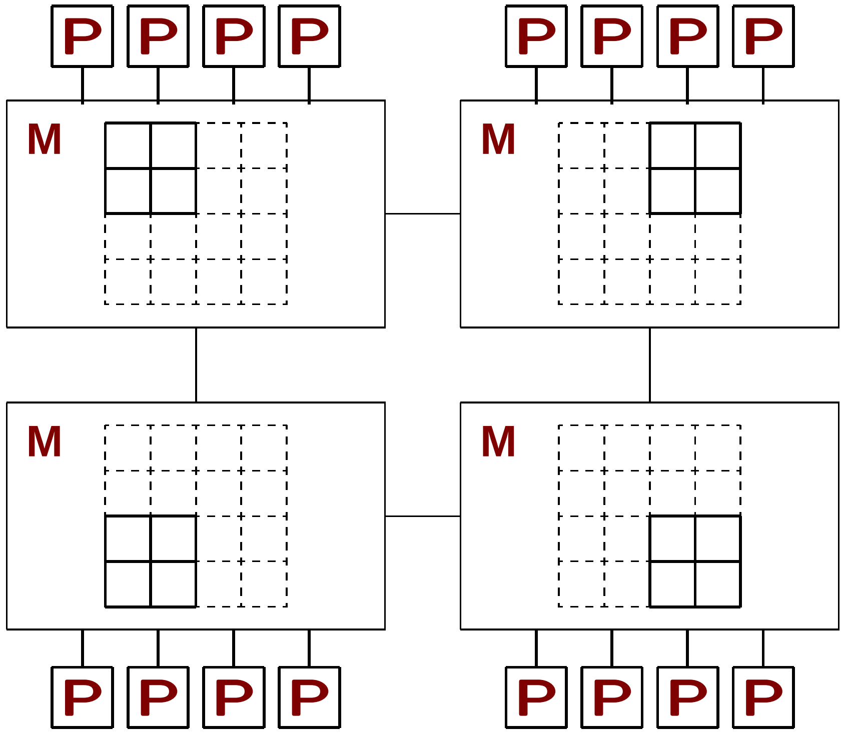
- shared across an mNode, distributed among mNodes.
- fewer and larger messages than distributed memory (communication/computation scales as surface/volume), may be less latency-bound.

Pure shared memory



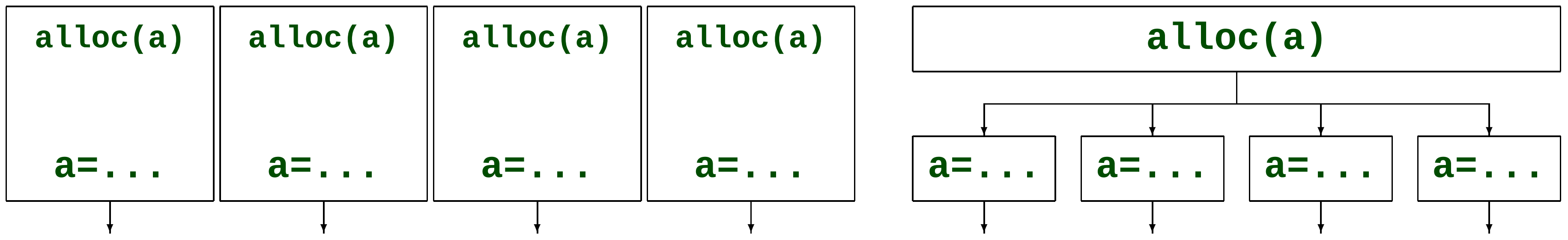
Array is local to one mNode: other mNodes requires remote loads and stores. OK on platforms that are well-balanced in bandwidth and latency for local and remote accesses. **ccNUMA** ensures cache coherence across the aNode.

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 affinity.

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 (more on that later).

Coding standards: MPI

`MPI` is the coding standard for message-passing and RMA; `shmem` is the non-standard.

- `MPI`/`shmem` are subroutine libraries to be ***called***, which we've wrapped in the MPP modules:

```
real, dimension(is-1:ie+1,js-1:je+1) :; a, b
type(domain2D) ::  domain
...
call mpp_update_domains( domain, a )
do j = js,je
   do i = is,ie
      a(i,j) = 0.5*( b(i+1,j)+b(i-1,j) )
   end do
end do
```

(2)

Coding standards: OpenMP

OpenMP is the standard for shared memory.

- **OpenMP** is ***directive***-based:

```
real, dimension(is:ie,js:je) ::  a, b
...
!$OMP parallel private(i,j) shared(a,b)
do j = js,je
   do i = is,ie
      a(i,j) = 0.5*( b(i+1,j)+b(i-1,j) )
   end do
end do
```

(3)

The directives are processed by the compiler to generate the appropriate synchronization and communication code.

Advantages of shared memory directives

- Directives are easy to apply to serial code.
- If the parallel axis changes often: e.g in 2D spherical harmonics (Fourier transform has global dependencies in X, Legendre transform has global dependencies in Y). The distributed memory implementation of spherical harmonics requires a parallel transpose for each transform operation.
- Block structured arrays:

```
real ::  a(nx,ny,nz,nblock)
...
!$OMP parallel private(n) shared(a)
do n = 1,nblock
   call foo( a(:,:,:,n) )
end do
```
(4)

Disadvantages of shared memory directives

- There is a cost associated with thread synchronization at the start and end of an OMP region: it generally works best with one large parallel region (***macrotasking***) as opposed to loop-level (***microtasking***).

- Reliance on parallel code generated by a compiler.

- Necessity to write ***thread-safe code***: care is needed in to weed out hidden and uninentional sharing, or having two copies of a variable you meant to share.

- ***False sharing*** is another issue: since memory loads are done in units of cache lines (4 or 8 words), processors working on neighboring words of a shared array can generate cache thrashing.

- Only limited shared memory is available on most platforms. However, the Altix being a notable exception, this does not apply to us.

In sum, shared memory is best used for ***extending scalability*** once a code has hit its scaling limit under message-passing. We are almost certain to need hybrid-parallel codes to improve scaling on current and near-future machines.

A caveat about hybrid OpenMP/MPI on Altix: OpenMP belongs to the compiler (Intel) and MPI to the MPT Toolkit (SGI)...

Extending scalability: AM2-FV

The AM2 suite of models using the FV core are a prime candidate for scalability extension.

- The FV core uses distributed memory in the Y direction only: key operations such as the polar filter (global X dependency) and vertical mapping (global Z dependency) inhibit distribution in the other directions. With a minimum of 3 latitude rows per processor, we are limited to 30 PETs at M45 resolution, 60 PETs at M90.

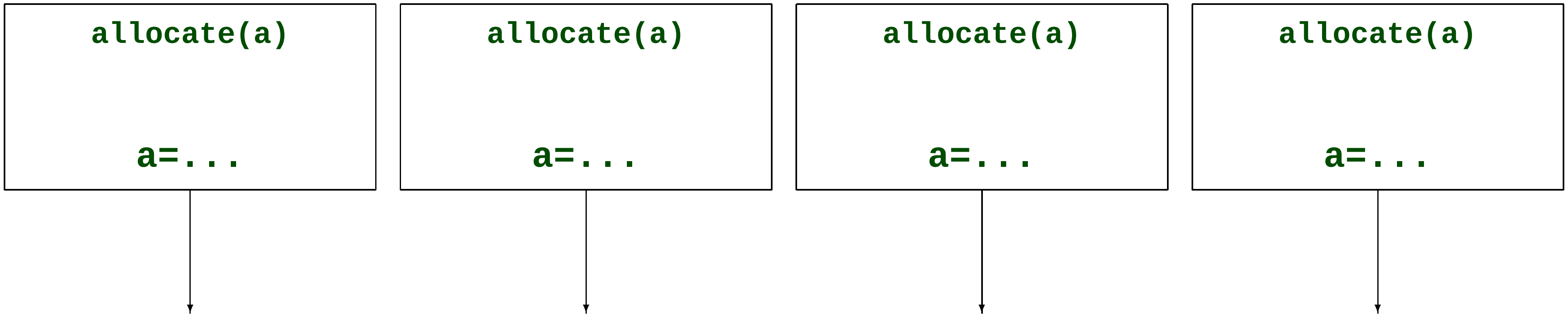
- The FV core uses OpenMP to extend scalability: the core alone can generate 3.5X speedup on 4 OMP threads.

- The AM2 physics component can be decomposed in 2D (there are only Z dependencies) but is not written to be thread-safe. The FMS manual had recommendations on how to write thread-safe code: all **`alloc`**s and **`dealloc`**s in the **`_init`** and **`_end`** routines, isolate I/O from compute, etc.: these were never enforced.

- Attempts to convert the AM2 code to thread-safe did not bear fruit: internal accumulators and alarms, allocated temporaries, module globals that all threads needed to update, these were some of the culprits. The kernel-driver approach currently being developed should address some of these issues.

A new approach to shared memory across PETs

The roots of the new approach lie in the **shmem** library concept of ***symmetric memory*** developed by Numrich et al.



```
call shmem_get( a, 1, pet+1 )
```
(5)

How does a PET know the address of a variable on another PET? In ***symmetric memory***, the numerical value of the address of any static variable is the same on all PETs.

On the Origin, this was extended to the idea of a ***symmetric heap***: even dynamically allocated variables on the symmetric heap could be addressed by any PET. (Caveat: this only works for heap, not stack: automatic arrays and other stack variables could not be remotely addressed).

On Altix, SGI could not build a symmetric heap: the ***malloc*** call on which this is based belongs to (Intel's) system library. But more on that later...

PSETs: Persistent Shared-memory Execution Threads

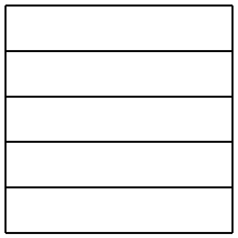
Based on the idea of symmetric shared arrays, we can exploit shared memory within PETs. This is what we call ***PSETs: Persistent Shared-memory Execution Threads***. Recall that TETs are created and destroyed after the declaration of shared variables. PSETs are persistent from the beginning to end of a job launched by `mpirun`. The array is declared by the “root PSET” and its address broadcast to all PSETs sharing it.

```
integer ::  n
real, allocatable ::  a(:)
real ::  b(n)
pointer( ptr, b )
...
if( root_pset )then
   allocate( a(n) )
   ptr = LOC(n)
end if
call mpp_broadcast_ptr( ptr, root_pset )
...
...  = b(i) ...
```
(6)

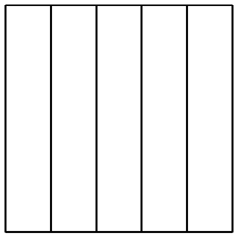
Except for the pointers that have been explicitly shared by this mechanism, nothing is required to be thread-safe.

Shared memory barriers

Synchronization of shared arrays is implicit in OpenMP and explicit in PSETs.



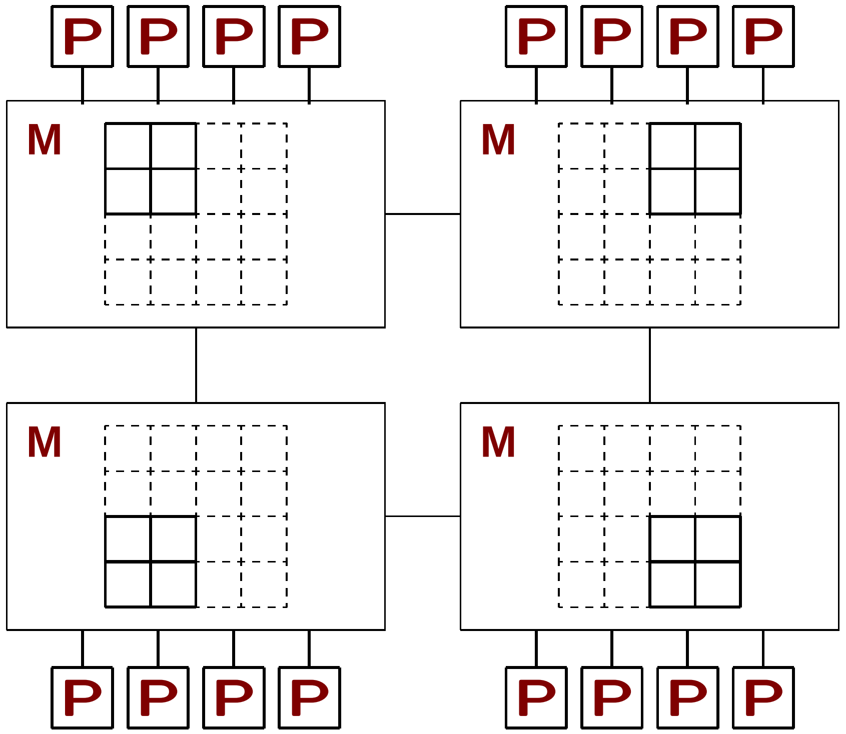
sync()



```
!$OMP parallel private(i,j) shared(a)
do j = js,je
   do i = is,ie
      a(i,j) = ...  a(i+1,j) ...
   end do
end do
call fv_array_sync()
do j = js,je
!$OMP parallel private(i,j) shared(a)
   do i = is,ie
      a(i,j) = ...  a(i,j+1) ...
   end do
end do
```

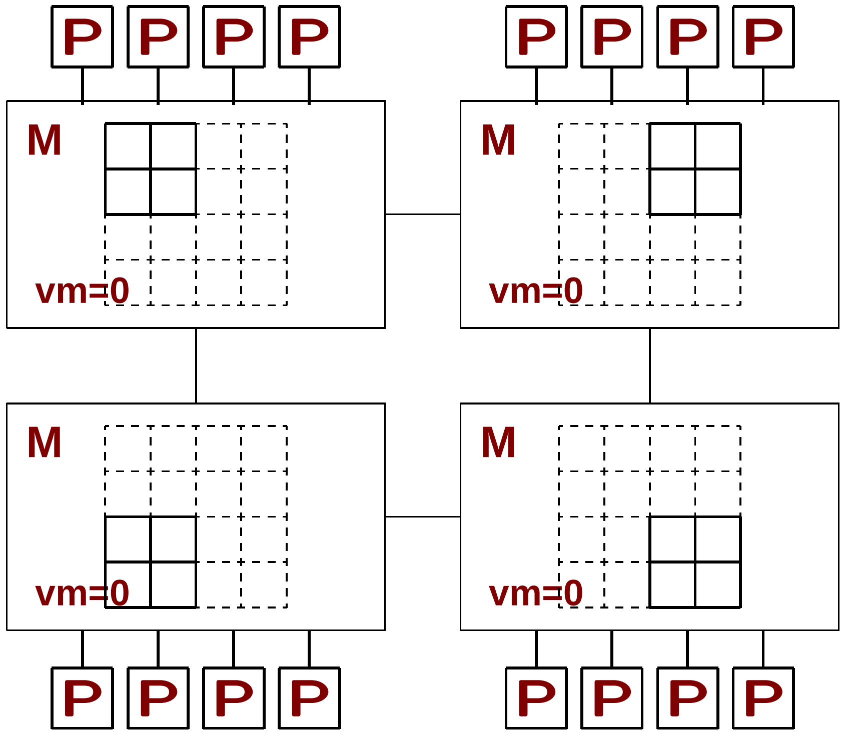
(7)

Intelligent memory allocation on PSETs



Processor-memory affinity and memory locality may be achieved by using `first-touch` heuristic.

PSETs on Altix

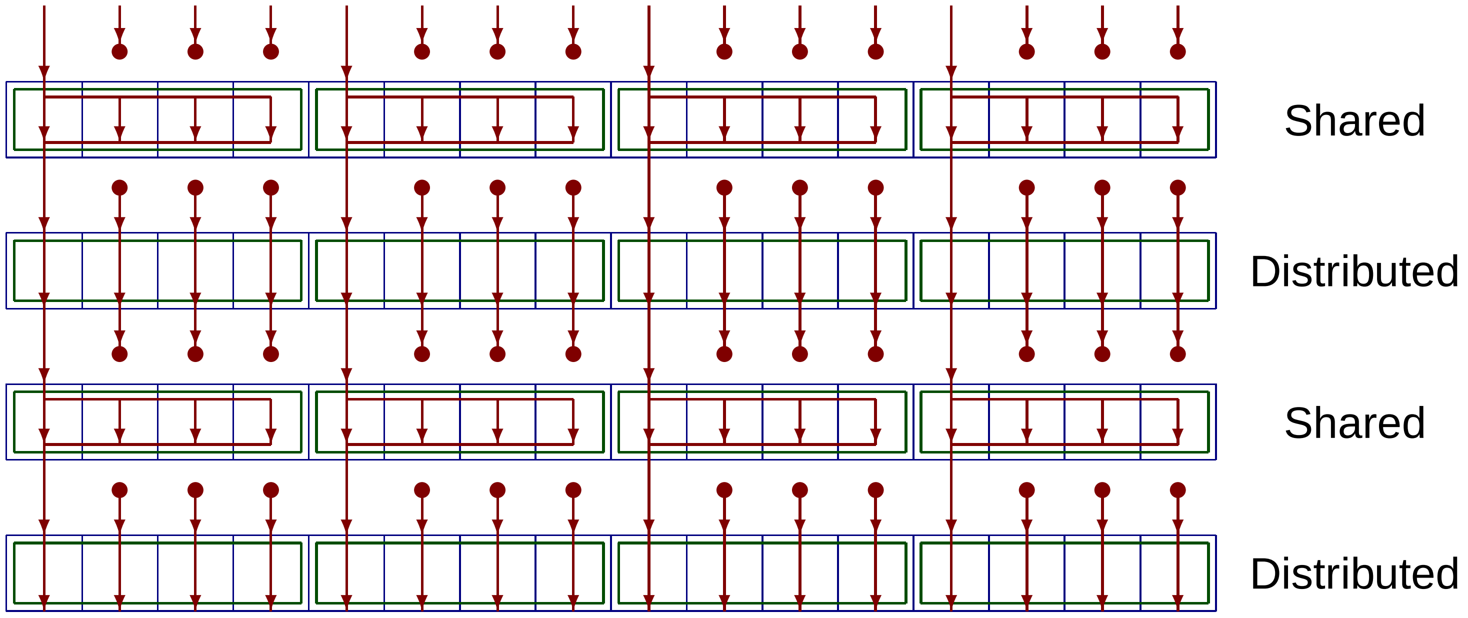


SGI provided a special address translation routine **`MPI_SGI_GlobalPtr`** to translate virtual memory addresses between PSETs.

Stack and automatic arrays are also supported through a special MPP implementation called **`mpp_pset_stack`**.

First attempt: Overlaying PSETs and TETs

The first attempt at using PSETs on AM2-FV attempted to do minimum violence to the FV core. It continued to use its OMP directives, but when the code entered the AM2 physics, it used PSETs instead, and the FV arrays became shared arrays. This was called the FV1D2D model.



Failure of FV1D2D

This should have worked! but I was never able to get the TETs to appear on the same processors as the sleeping PSETs. That is, if I launched this jobs as:

- **`mpirun -np 16 FMS.x`**

  it ran very slowly (thread contention), but if I ran the same as

- **`mpirun -np 28 FMS.x`**

  I got near-perfect 16X scaling.

There is clearly much more to learn about the control and scheduling of OpenMP threads...

Second attempt: a complete implementation of AM2-FV using PSETs

The current version of the FV core (now with the `testing` tag, soon to be `memphis`) is now implemented completely using PSETs.

- PSETs used within the FV core itself instead of OpenMP (OMP directives still in place, but not used).
- FV continues to use distributed memory in the Y direction.
- Runs correctly and bitwise-exactly on multiple threads in Held-Suarez, AM2, and concurrent coupled modes.
- Does not match Lima answers because of algorithmic changes. Lima answers can be matched by using flag `-DUSE_LIMA`, but that version does not have PSETs.
- Scales well on 2 threads, disappointing falloff on 3 and 6 threads (`npset` must be a divisor of 6 for M45L24). But we already expect to be able to run CM2.1 at a minimum of 15 years/day (including gains from the compiler). The exact load-balanced processor count for coupled models is still being worked out.

PSETs are still a work in progress...

- We are still trying to understand and improve its performance. Issues include PSET barrier optimization, compiler settings, memory locality, physics “window” PSETs, ...
- PSETs can be of great utility in other codes: ocean explicit free surface, spectral transforms, ...
- Similar functionality is being explored on other platforms (Cray, IBM).
- The final goal is to achieve a ***uniform memory model*** (see Balaji and Numrich 2005), where users will be able to address distributed arrays without caring too much whether the underlying memory semantics are distributed, shared, or hybrid.
- Many thanks for technical advice and enthusiastic participation from Gerardo Cisneros, Jeff Durachta, Karl Feind, Koushik Ghosh, Kim McMahon, Alex Pletzer, and of course above all, S.J Lin...