Memory Coherence in Shared Virtual Memory Systems  
Kai Li and Paul Hudak  
1986

Why is this cool?  
All the advantages of *very* expensive parallel hardware.  
On cheap network of workstations.  
No h/w modifications required!

What's the basic claim?

What's a loosely coupled multi-processor?  
A bunch of workstations on a LAN.  
Background: shared-memory multiprocessors.

What's the basic architecture?  
Workstations on a LAN.  
Split up processing among the workstations.  
Have them be able to read/write each other's memory.

Typical s/w arrangement:  
For parallel sort.  
Load array into memory.  
Each host responsible for a piece of the array.  
On host $i$:  
  Do a local sort on its array piece.  
  $\text{done}[i] = true$;  
  Wait for all $\text{done}[]$ to be true.  
  Now merge my piece with neighbors...

Let's work out a design for ourselves.

Do we want a single address space?  
Or have programmer understand remote references?

Shall we make a fixed partition of address space?  
I.e. first megabyte on host 0, second megabyte on host 1, &c?  
And send all reads/writes to the correct host?  
We can detect reads and writes using VM hardware.  
I.e. I read- or write-protect remote pages.  
What if we didn't do a good job of placing the pages on hosts?  
Maybe we cannot predict which hosts will use which pages?

Could move the page each time it is used.  
When I read or write, I find current owner, and I take the page.  
So need a more dynamic way to find current location of the page.  
What if lots of people read a page?

Move page for writing, but allow read-only copies.  
When I write a page, I invalidate r/o cached copies.  
When I read a non-cached page, I find most recent writer.  
Works if pages are r/o and shared, or r/w by one host.  
Only bad case is write sharing.  
When might this arise? False sharing...

Does this have the right semantics?  
I.e. will memory read/write behave as programmers expect?  
Does it work for our compute ; $\text{done}[i] = true$ example?  
No stale data. Relative order.  
Does it work for Lamport's mutual exclusion algorithm?  
(Fast Mutual Exclusion paper)  
$x := i$;  
if $y != 0$ then We Lose end;  
y := $i$;  
if $x == i$ then We Win end;  
Simultaneous writes to $x$; everyone must agree.

What do they claim about semantics.
They define "coherent".
Their rule: read sees most recent write to that address.
Is this the same rule that we've been using?
  What about simultaneous writes?
  What about order of writes to different addresses?

How about their constraint (start of Section 2).
  Only update while no other processing is using data.
  What does that mean?
  Why does it enforce coherence?

Let's talk about their basic algorithm, section 3.1.

per-page tables:
  manager's info[]: owner, copy_set, lock
  host's ptable[]: access, lock

Invariants?
  Every page has exactly one owner.
    It may have r/o copy...
      Only manager has to know who owner is, owner may not know.
    If owner has r/w, there are no other copies.
      copy_set must exactly match non-owner hosts w/ copies.

Five cases:
  I read a page I own or have cached read-only.
  I read a page I don't have.
  I write a page I own.
  I write a page I have cached read-only.
  I write a page I don't have.

Why does manager demand confirmation when handing out read copies?
  Page 232, Read server.
  Ensure atomic update of copy_set and reading host's access field.
  Otherwise data may reach host after owner changes hands, copy_set nulled.
  E.g. if simultaneous read and write.

Why does manager demand confirmation when changing owner?
  Page 232, Write Server.
  Before it gives up lock.
  Ensure atomic update of info[].owner and owner's access field.
  Otherwise 2nd write may overtake first, leaving two owners.
  E.g. if simultaneous writes.

What is the idea behind the improvement in 3.2?
  Put copy_set on owner, easier to update atomically, no confirmation.
  What if 2nd write overtakes first?
    Manager sends msg to the wrong owner.

What is the idea behind the distributed manager in 4.3?
  Don't try to track owner precisely.
    Every host keeps a hint per page pointing to last known owner.
    If that host isn't owner, it will forward us.

What's a block odd-even based merge-split algorithm?
  Why is appropriate to this kind of parallel architecture?
    Partition over N CPUS.
    Local sort at each CPU.
    View CPUS as logical chain.
    Repeat N times:
      Even CPUS send to (higher) odd CPUS.
      Odd CPUS merge, send low half back to even CPU.
      Odd CPUS send to (higher) even CPUS.
      Even CPUS merge, send low half back to odd CPU.
    Note that "send to" means look at the right place in shared memory.
      Probably everything in a single huge array.
What's the best we could hope for in terms of performance?
Linear speedup...

When are you likely to get linear speedup?
When there's not much sharing -- strict partition.

How well do they do?
Linear for PDE and matrix mul, not so good for sorting

Process memory vs O/S structures.
Can you see my open() file descriptors?

In what sense does it subsume the notion of RPC?
When would DSM be better than RPC?
More transparent. Easier to program.
When would RPC be better?
Isolation. Control over communication. Tolerate latency.
Portability.
Might you still want RPC in your DSM system? For efficient sleep/wakeup?

Has their idea been successful?
Using workstations for parallel processing: yes! Beowulf, Google, &c.
Shared memory? Hard to say. Lack of control over communication details?