TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems
Keleher, Cox, Dwarkadas, Zwaenepoel
USENIX 1994 Winter

Quick DSM tutorial
  (only if Ivy wasn't assigned)
  predecessor scheme for Treadmarks
  you have lots of workstations
  you want to use them in parallel for a single big computation
  you write your application using threads and locks
    maybe it's an existing threaded app, no changes required
  start a copy on each workstation
  use "distributed shared memory" so that they share all memory
  layering: app, DSM library, VM h/w, network
  VM h/w can mark page as invalid, read-only, or read-write
    fault to DSM library if you write r/o page, or anything an invalid page
  every page has a current owner
    page manager tracks current owner
    current owner required to have a copy!!! cannot toss it.
    management duties spread over workstations, page # mod N
  read fault (on an invalid page)
    ask manager who current owner is
    get a copy
    tell current owner to mark as read-only
    tell manager we have a copy
    mark as read-only on local machine
  write fault
    ask manager who current owner is
    get a copy (if not already present r/o)
    tell current owner to mark invalid, and give up ownership
    tell manager we're the owner
      manager tells all r/o copies to mark as invalid
    mark as r/w on local machine
  invariants:
    1. every page has exactly one current owner
    2. current owner has a copy of the page
    3. if mapped r/w by owner, no other copies
    4. if mapped r/o by owner, maybe identical other r/o copies
    5. manager knows about all copies
  sequentially consistent
    rules cause writes to a page to proceed one at a time

Treadmarks high level goals?
  Better DSM performance.
  Run existing sequential consistent parallel code.

What specific problems with previous DSM are they trying to fix?
  false sharing of large pages
  only one writer of a page at a time
  write of irrelevant data may invalidate my page
  Is false sharing the only main problem Treadmarks is solving?

What are write diffs?
  And what is the point?

```
  Are they just like object/variable granularity rather than page
granularity?
  No sub-page invalidate, so you would need to send updates after each
store.
  Do diffs make sense by themselves? I.e. w/o RC or LRC?

What is release consistency?
  And what is the point?
  When would RC be more efficient than just write diffs?

What is lazy release consistency?
  And what is the point?
  When would LRC be more efficient than RC?

Why do they need vector timestamps?

Example for why you need the vector timestamps.
Assume each variable on its own page.
  Or that granularity is a single word...
CPU0: al1 x=1 rl1
CPU1:             al1 y=x rl1
CPU2:                         al1 print x, y rl1
What's the "right" answer?
How would an eager release consistent system handle this?
How does lazy release consistency handle this?
How does TreadMarks know what to do?

Example of why LRC might speed up your program.
CPU0: al1 x=1 rl1  al2 z=99 rl2
CPU1:              al1 y=x rl1
CPU2:                           al1 print x, y, z rl1
How would an eager release consistent system handle this?
What does Treadmarks do?
  And why is that faster?
Why is it legal for z to be out-of-date at CPU2?
How does Treadmarks *know* it is legal?

What if you get different values from different sources?
CPU0: al1 x=1 rl1  al2 y=9 rl2
CPU1:              al1 x=2 rl1
CPU2:                           al1 al2 z = x + y rl2 rl1
CPU2 is going to hear "x=1" from CPU0, and "x=2" from CPU1.
  How does CPU2 know what to do?

What if the VTs for the two values are not ordered?
  Could this happen?
CPU0: al1 x=1 rl1
CPU1:              al2 x=2 rl2
CPU2:                           al1 al2 print x rl2 rl1

What model of consistency does the programmer need to have in mind?
  What rules does the programmer have to follow?
  What can the programmer expect from the memory system?

Example of when LRC might still do too much work.
CPU0: al2 z=99 rl2  al1 x=1 rl1
CPU1:                           al1 y=x rl1
```

```
In this case, CPU1 didn't really need z.
This suggests that further improvements might be possible,
  at what expense?
  Compiler or programmer notices dependencies?

What happens in this case?
CPU0: al1 x=1 rl1                      al1 x=2 rl1
CPU1:              al1 al2 y=x rl2 rl1
CPU2:                                              al2 z=y rl2
Does CPU2 get the x=2 update? Should it? Does it make any difference?
  In most cases TreadMarks could avoid sending x=2 if it wanted to.
  But there might have been a GC, forcing CPU2 to know about x=2.

Are there programs that work under RC that break under LRC?
  RC is blind to which lock variable was involved.
CPU0: al1 x=1 rl1  al1 if(y==0) ... rl1
CPU1: al2 y=1 rl2  al2 if(x==0) ... rl2

TreadMarks keeps complete history of write notices.
  I.e. accurate record of what changed in each interval.
  Write notice and interval record lists in Figure 2.
  It does not just keep the latest data.
  Why? Is the history needed for LRC?

What's the point of the diffs?

Why do they send write notices separately from the diffs?
  They can delay the diffs until you actually use the page: you might
not.
  Maybe you already have the write notice? But VT indicates this?

When you ask for diffs, how far back in the past do you ask for them?
  Does this imply unbounded amount of history?

Is LRC actually a good idea?
```