Shared Memory Architectures

Shared Memory Programming
Wait-Free Synchronization
Intro to SW Coherence

Discuss paper on Ultracomputer

Agarwal

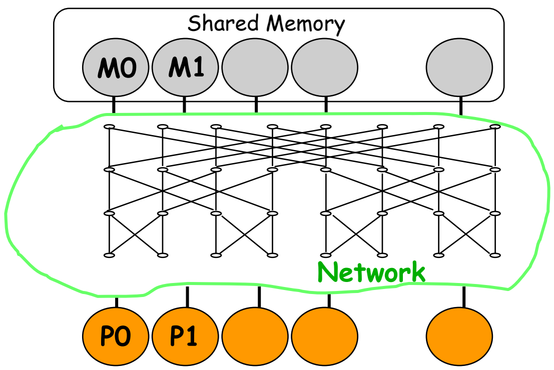
6.173
Fall 2010
L08

Today's Outline

- Shared memory programming
- Dynamic load balancing and work Qs
  - Jacobi
  - TSP
- Ultracomputer/RP3 discussion
  - Shared memory machines
- Wait-free synchronization
- How do caches change things
- Software coherence

Ultracomputer Design

Discuss



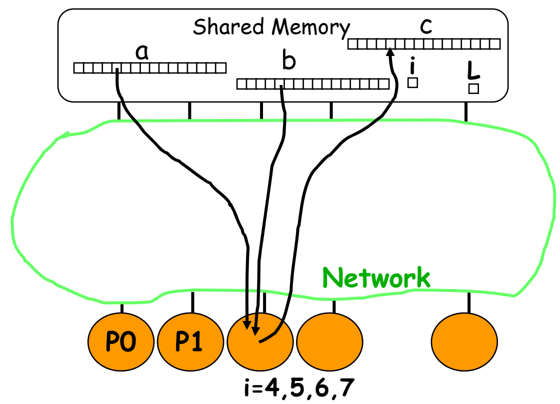
This is a butterfly network – variant of Omega network

- Indirect network – Omega network (details later in course)
- Shared memory machine
- Communication/synchronization through shared memory
- Hardware routing of memory requests
- No latency hiding – wait for memory request
- What were the big ideas?

Concept built as IBM RP3 machine (we will see this later)

Pure Shared Memory

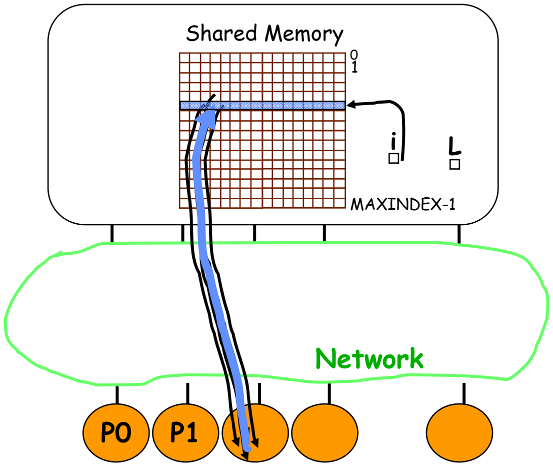
So, getting a work item is not so cheap after all, is it? Any ideas?



Coarse grain parallelism (versus fine grain parallelism): Get a block of 4 or 16 or more indices each time to amortize the overhead of locking

Comment: Index i is like an implicit work Q

Jacobi
Let's Try Same Basic Concept



Getwork() grabs an index to a row (e.g.)
Needs synchronization as before
Perform loads and stores on shared array

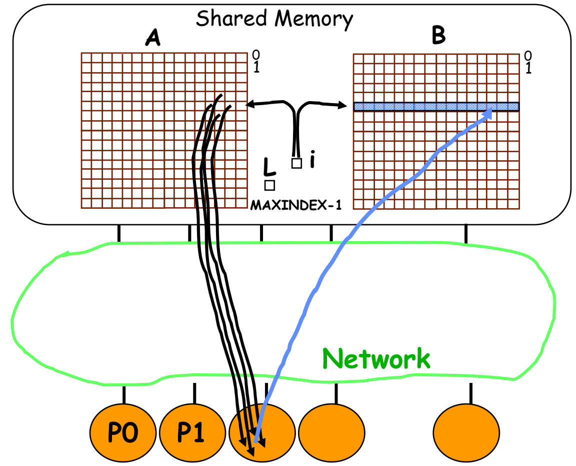
Does this work? Or, is result different from the msg version

Behavior is different from message passing version 'cos updates are in "in-place" (others "see" my updates); But still ok in terms of physics (we will discuss alternate way next)

OK, so finish row. Then try to grab another index, until index reaches MAXINDEX

Or, to do more iterations of jacobi, index can roll over to 0

Double Buffering Idea
A Common Trick in Shared Memory Programming



Getwork() grabs an index to a row
Needs synchronization as before
Perform loads on shared array A and store into shared array B

Finish assigned row. Then try to grab another index, until index reaches MAXINDEX. Iteration is now complete

In the next iteration, we will read from B and write into A

But, how do I know I can start the next iteration?

Barrier. If a processor sees i to be MAXINDEX, it enters barrier

Also, need some care in resetting i

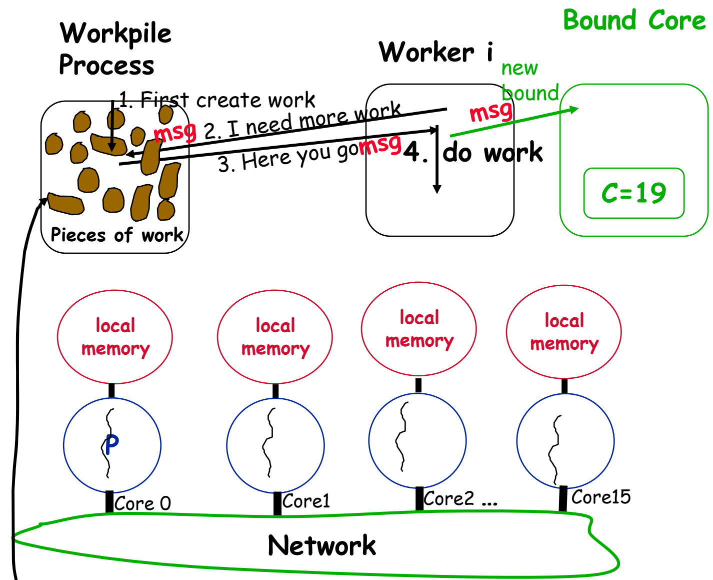
Load Balancing

Both our examples (adding vectors and jacobi) used dynamic load balancing

- **Each processor dynamically asked for a new piece of work (index) when it was done with its previous piece**
- **Incrementing a simple index to obtain work is simple and powerful**
- **A variant can be used even when the chunks of work are more complex; e.g., TSP**

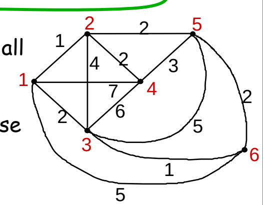
We can also use static load balancing where the programmer pre-assigns given index ranges to each processor

Recall Dynamic Load Balancing in Message Passing for TSP

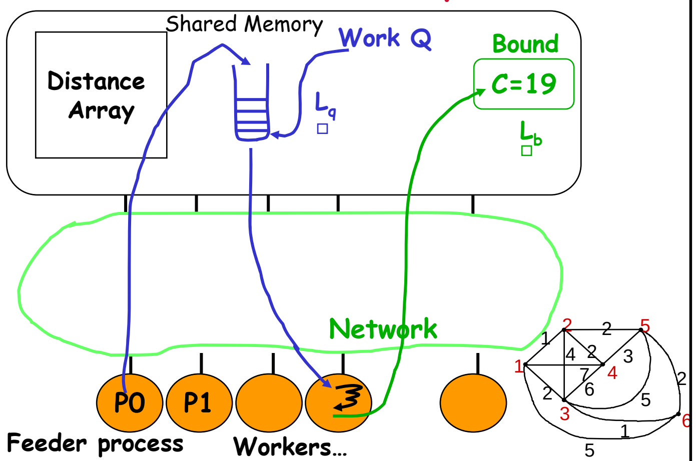


Starting from a given city, enumerate all partial paths of length N

Each worker core is passed one of these partial paths and fleshes it out (bound updated as new bounds are found)



TSP in Shared Memory



Starting from a given city, one feeder process can enumerate all partial paths of length L and put them on work Q (SPMD?)

Then, workers call Getwork(), which accesses work Q and returns a work item; workers look for tours, updating bound as usual

Work Q is key concurrent data structure

- Simplest approach is to use "index" into a work array
- Index increment must be atomic as in Jacobi

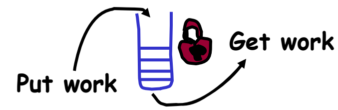
If work Q supports atomic puts and gets, feeder and workers can run in parallel! Can also build hierarchical work Qs, …

(Talk to Eastep about Smart Work Qs!)

Work Qs

Single work Q with sequential static work creation – can use simple index with atomic increment for gets

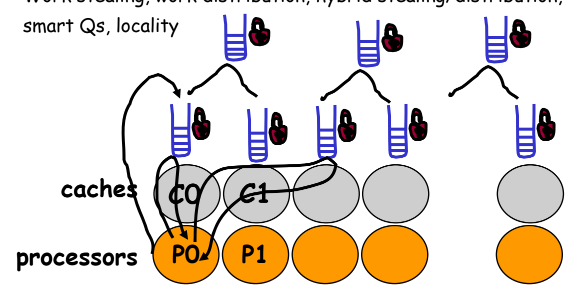
Single work Q with dynamic work creation – need atomic puts and gets



Work Q is often bottleneck in dynamic work creation situations

System throughput often limited by work Q throughput

Discuss ideas for clever work Qs:
Work stealing, work distribution, hybrid stealing/distribution, hierarchy, smart Qs, locality



Synchronization in Shared Memory

We have seen many examples

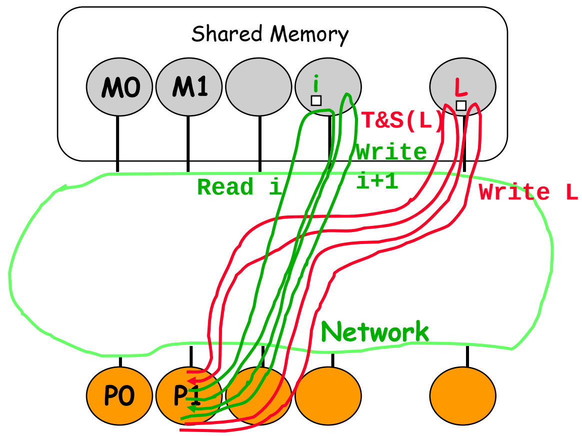
Barrier
- Everyone done with given step?

Locks
- Mutual exclusion
- Increment index (work item)
- Update counter (global bound)
- Atomically access work Q

In many cases the critical section was short, e.g., increment index in getwork

Locks seem like a wasteful approach for such short computations

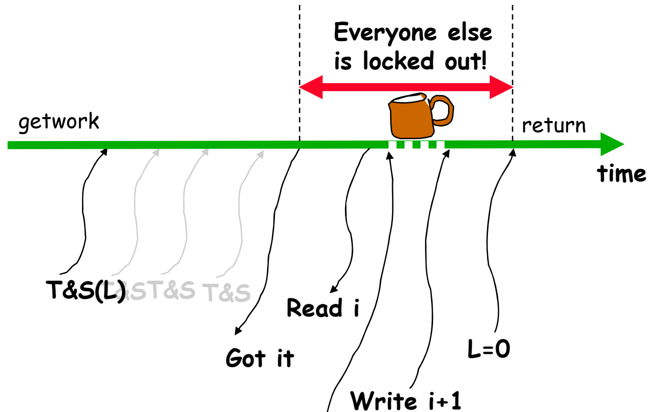
Locks are Expensive
Network Traffic




```
int getwork()
{
        while (T&S(L) == 1) {}; /* get lock */
        i = i + 1; /* increment index */
        L = 0; /* release lock */
        return(i);
}
```

Locks are Expensive

Serialization Bottleneck

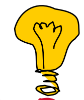



```
int getwork()
{
        while (T&S(L) == 1) {}; /* get lock */
        i = i + 1; /* increment index */
        L = 0; /* release lock */
        return(i);
}
```

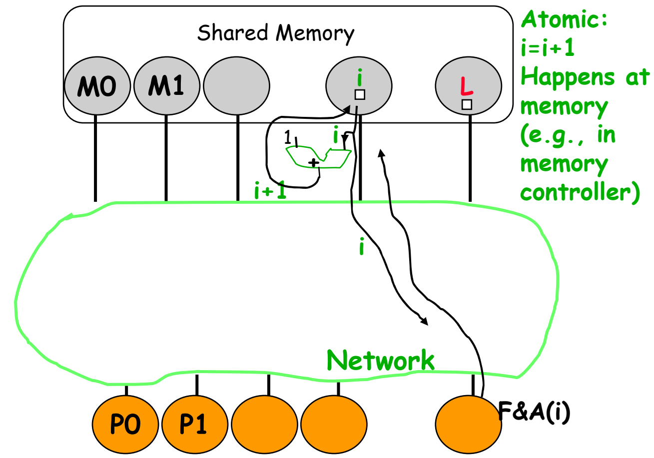
"increment i" throughput limits throughput of the system

Worse, imagine if the lock holder takes a coffee break!

Can we do better?



Concept of Wait-Free Synchronization



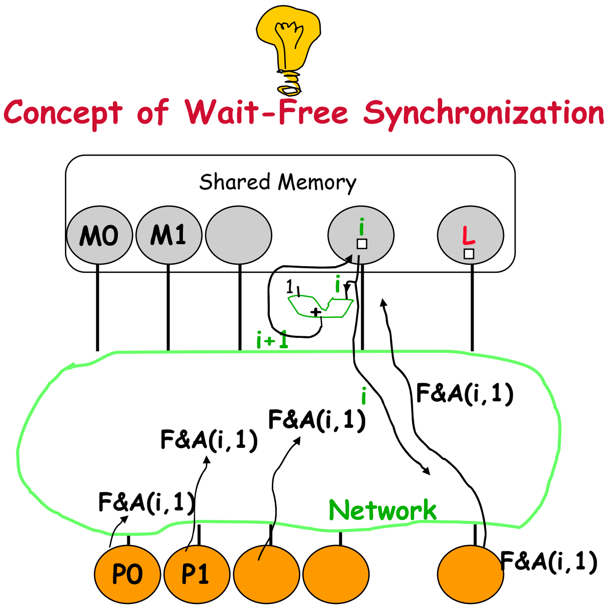
Atomic: i=i+1 Happens at memory (e.g., in memory controller)

```
int getwork()
{
        i = F&A(i,1);
        return(i);
}
```

Wiki Fetch-and-Add:
« atomic »
function FetchAndAdd(*address* location) {
    int value := *location
    *location := value + 1
    return value
}

```
int getwork()
{
        while (T&S(L) == 1) {}; /* get lock */
        i = i + 1; /* increment index */
        L = 0; /* release lock */
        return(i);
}
```

Concept of Wait-Free Synchronization

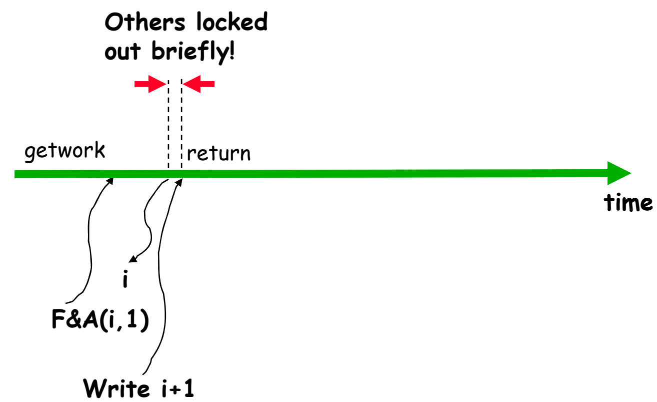



```
int getwork()
{
        i = F&A(i,1);
        return(i);
}
```

F&A ops queue up in the network or in the memory controller and are handled quickly as they arrive

F&A Minimizes Serialization

Wait-Free Synchronization




```
int getwork()
{
        i = F&A(i,1);
        return(i);
}
```

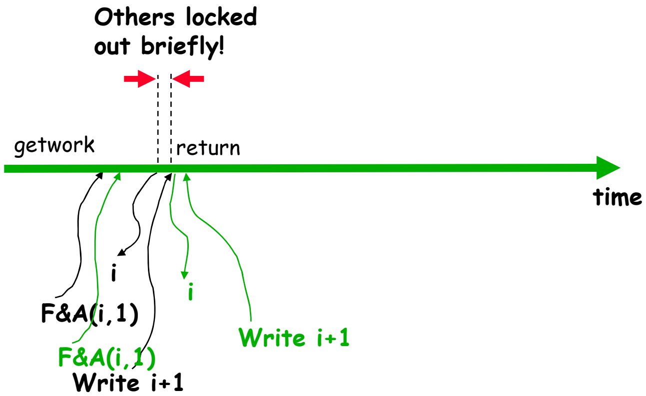
"increment i" is fast, so improves throughput of the system

Coffee break by a process only hurts that process!

Many other such ops possible: F&Xor etc.

F&A Minimizes Serialization

Exploits more parallelism

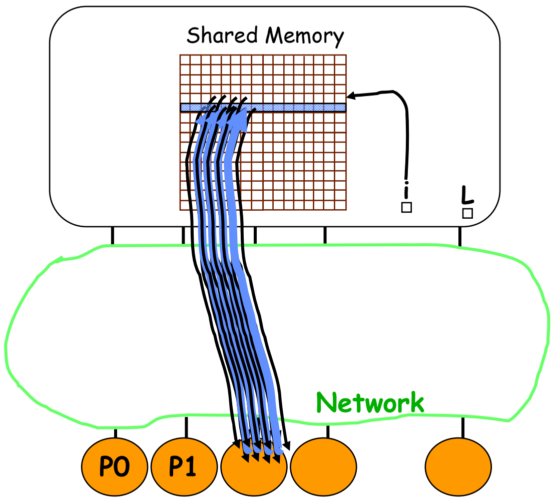



```
int getwork()
{
        i = F&A(i,1);
        return(i);
}
```

"increment i" is fast, so improves throughput of the system

We will look at more fun stuff with F&A and relatives (e.g., cmp&swap) and their cool implementations later in the course

Back to Shared Memory Jacobi



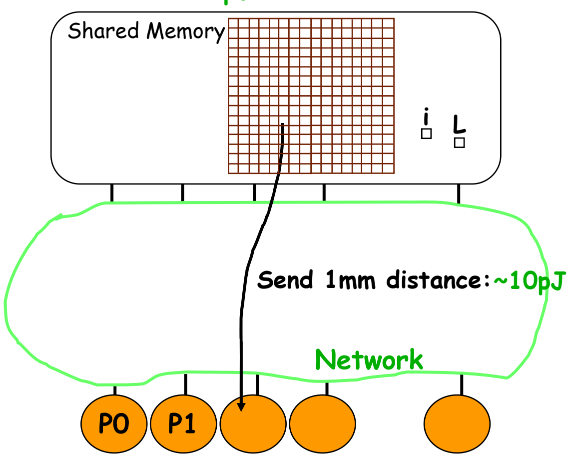
Lots of repeat accesses of data
Lots of communication over the network

And very energy inefficient

Pure Shared Memory

32-bit energy costs in 40nm

DRAM read: ~1000pJ



Add: ~ 1pJ

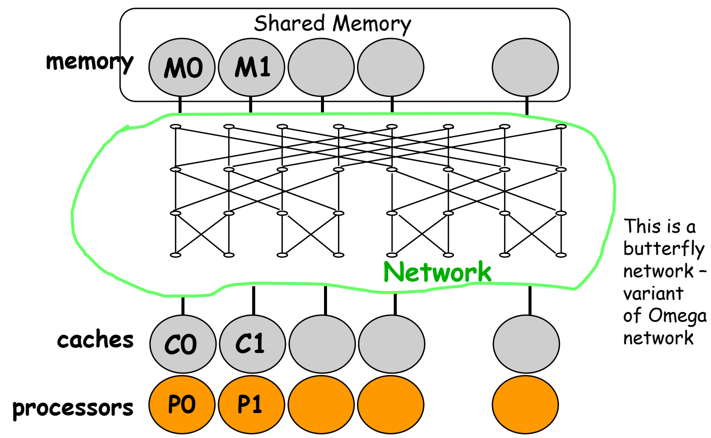
Register read: ~1pJ

Ideas?

Caches!

Cache read (small L1): ~10pJ

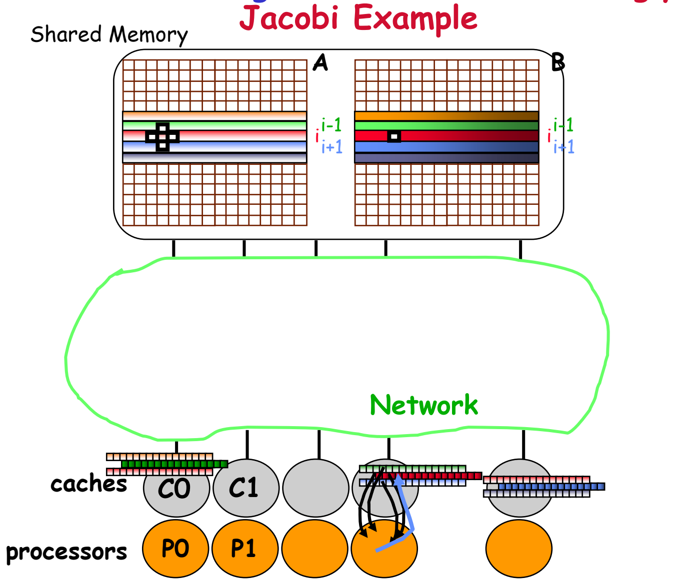
RP3



- Indirect network – Omega network (details later in course)
- Shared memory machine with caches
- Each memory module placed physically close to processors
- Communication/synchronization through shared memory
- Hardware routing of memory requests
- SPMD FORTRAN programming (single program multiple data)
- No latency hiding – wait for memory request
- More complex hardware
- I could not find a picture of the RP3

What were the big ideas?

Caches – the good, the bad and the ugly
Jacobi Example

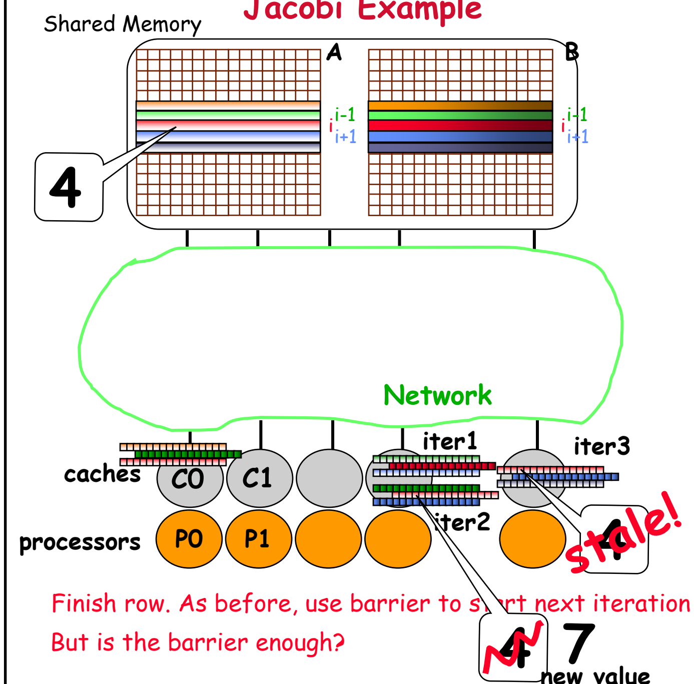


Caches exploit spatial locality here (fetch cache line)
Temporal locality too (discuss)
Network traffic dramatically reduced! Lower energy too

Finish row. As before, use barrier to start next iteration

But is the barrier enough?

Caches – the good, the bad and the ugly
Jacobi Example



Finish row. As before, use barrier to start next iteration

But is the barrier enough?

Cache coherence problem! What do we do?

Maintaining coherence in manycores Major approaches

- User-software managed coherence
  - RP3
  - Beehive
- System-software managed coherence
- Hardware managed coherence (later in the course)

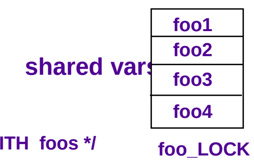
User-software managed coherence in manycores

Typically yields weak coherence

i.e. Coherence at sync points (or fence pts)

E.g.: When using locks for shared object accesses

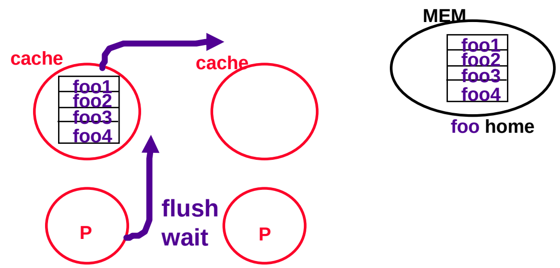
Code:




```
GET_foo_LOCK
    /* MUNGE WITH  foos */
    foo1 =
    X = foo2
    foo3 = .
           .
           .
RELEASE_foo_LOCK
```

How do you make this work?

User Software Coherence



GET_foo_LOCK

.
. MUNGE
.
.

Flush foo* from cache, wait till done

RELEASE_foo_LOCK

Issues ...the ugly

- Need special processor instructions for flush (e.g., beehive)
- Also need a "memory fence" (wait till all flushed local values are reflected in global store)... next
- Can you cache the lock?
- Must be conservative; when in doubt, flush
  - Lose some locality
- But, can exploit application characteristics to allow some inconsistency

  e.g. TSP – bound does not have to be accurate

  Chaotic relaxation **to be continued ...**

- 25 -