More Advanced OpenMP

- This is an abbreviated form of Tim Mattson’s and Larry Meadow’s (both at Intel) SC ’08 tutorial located at http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

- All errors are my responsibility

Topics (only OpenMP 3 in these slides)

- Creating Threads
- Synchronization
- Runtime library calls
- Data environment
- Scheduling for and sections
- Memory Model
- OpenMP 3.0 and Tasks

OpenMP 4

- Extensions to tasking
- User defined reduction operators
- Construct cancellation
- Portable SIMD directives
- Thread affinity

Creating T*asks*

- We already know about
  - parallel regions (*omp parallel*)
  - parallel sections (*omp parallel sections*)
  - parallel for (*omp parallel for*) or *omp for* when in a parallel region
- We will now talk about Tasks

Tasks

- OpenMP before OpenMP 3.0 has always had tasks
  - A parallel construct created *implicit* tasks, one per thread
  - A team of threads was created to execute the tasks
  - Each thread in the team is assigned (and *tied*) to one task
  - Barrier holds the original master thread until all tasks are finished (note that the master may also execute a task)

Tasks

- OpenMP 3.0 allows us to *explicitly* create tasks.
- Every part of an OpenMP program is part of some task, with the master task executing the program even if there is no explicit task

task construct syntax

#pragma omp task [clause[[,]clause] **...**]
structured-block

clauses:
if (expression**)**
untied
shared (list**)**
private (list**)**
firstprivate (list**)**
default(shared | **none)**

Blue options are as before and associated with whether storage is shared or private

if (false) says execute the task by the spawning thread

- different task with respect to synchronization
- Data environment is local to the thread
- User optimization for cache affinity and cost of executing on a different thread

untied says the task can be executed by more than one thread, i.e., different threads execute different parts of the task

When do we know a task is finished?

- At explicit or implicit thread barriers
  - All tasks generated in the current parallel region are finished when the barrier for that parallel region finishes
  - Matches what you expect, i.e., when a barrier is reached the work preceding the barrier is finished
- At task barriers
  - Wait until all tasks defined in the current task are finished

```
#pragma omp taskwait
```

  - Applies to tasks *T* directly generated in the current task, not to tasks generated by the tasks *T*

Example: parallel pointer chasing with parallel region

```
#pragma omp parallel
{
  #pragma omp single private(p)
  {
    p = listhead ;
    while (p) {
      #pragma omp task
          workfct (p)
      p=next (p) ;
    }
  }
}
```

value of *p* passed is value of *p* at the time of the invocation. Saved on the stack like with any function call

workfct is an ordinary user function.

Example: parallel pointer chasing with for

```
#pragma omp parallel
{
  #pragma omp for private(p)
  for ( int i =0; i <numlists ; i++) {
    p = listheads [ i ] ;
    while (p ) {
      #pragma omp task
              workfct (p)
      p=next (p ) ;
    }
  }
}
```

Example: parallel postorder graph traversal

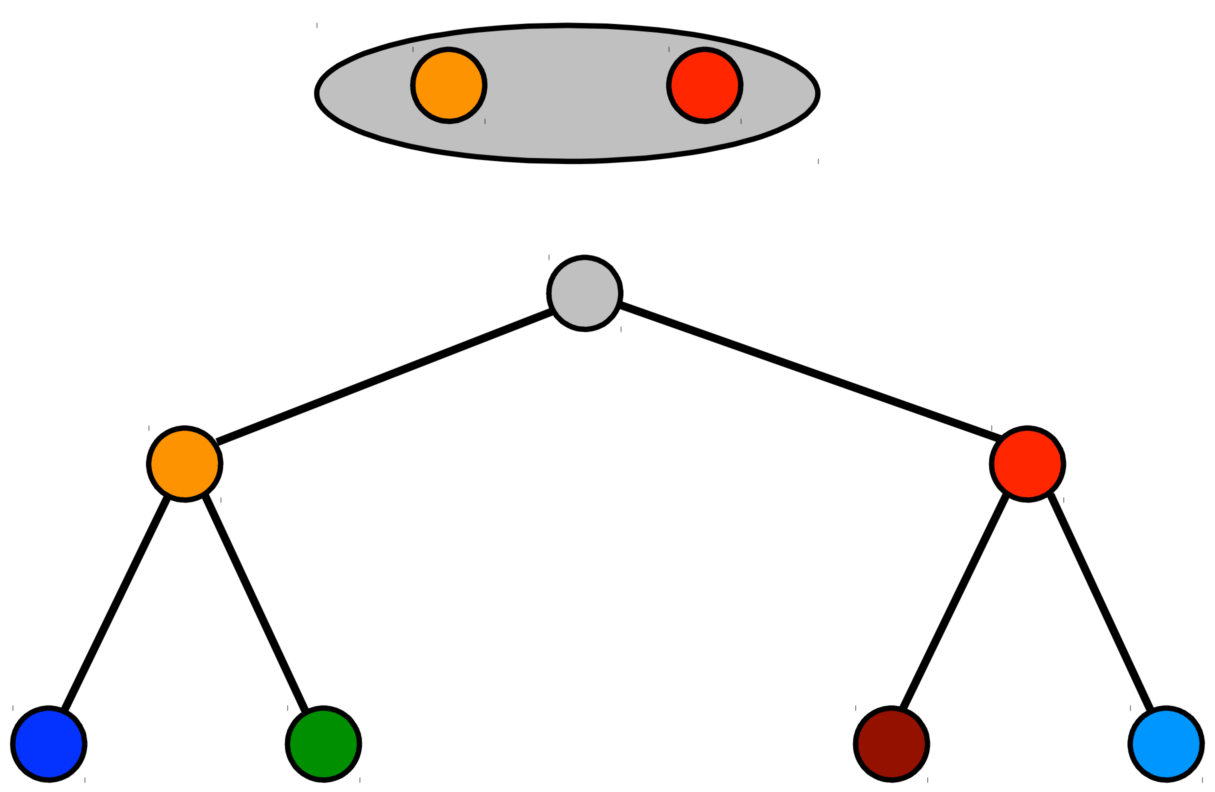
```
void postorder(node *p) {
   if (p->left)
      #pragma omp task
            postorder(p->left);
   if (p->right)
      #pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for descendants
   workfct(p->data);
}
```

Parent task suspended until child tasks finish

This is a task scheduling point

Example: postorder graph traversal in parallel

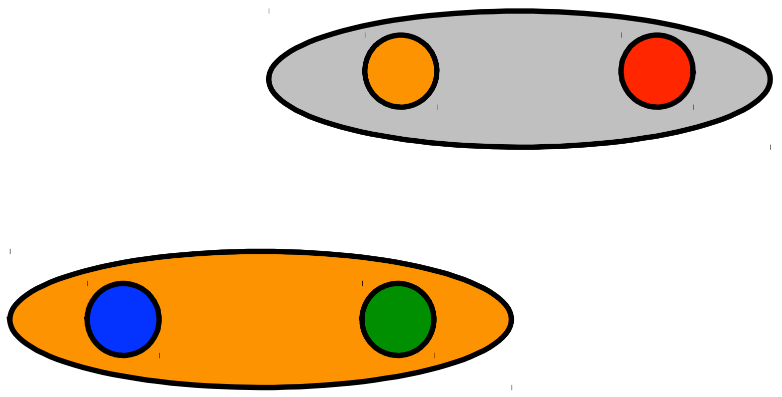
```
void postorder(node *p) { // p is ● initially
   if (p->left)
      #pragma omp task
            postorder(p->left);
   if (p->right)
      #pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for descendants
   workfct(p->data);
}
```

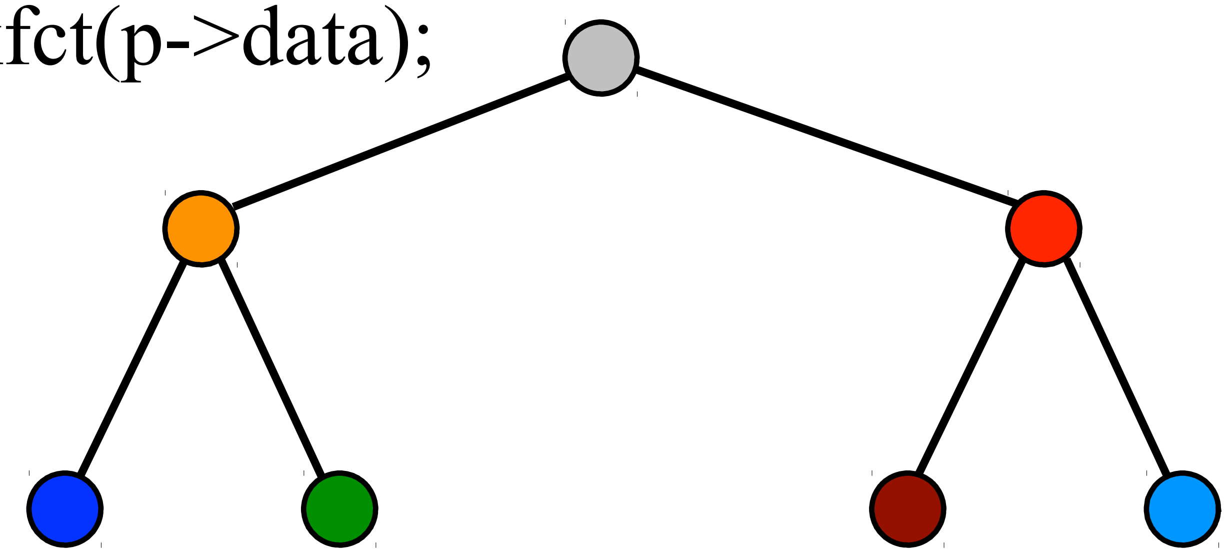


Postorder is called from within an omp parallel region

Postorder graph traversal in parallel — task wait

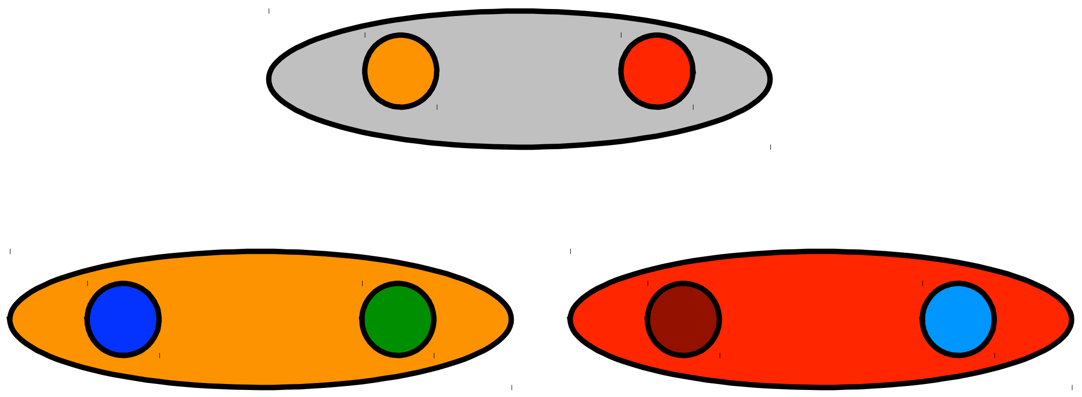
```
void postorder(node *p) { // p is ●
   if (p->left)
      #pragma omp task
            postorder(p->left);
   if (p->right)
      #pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for
descendants
   workfct(p->data);
}
```

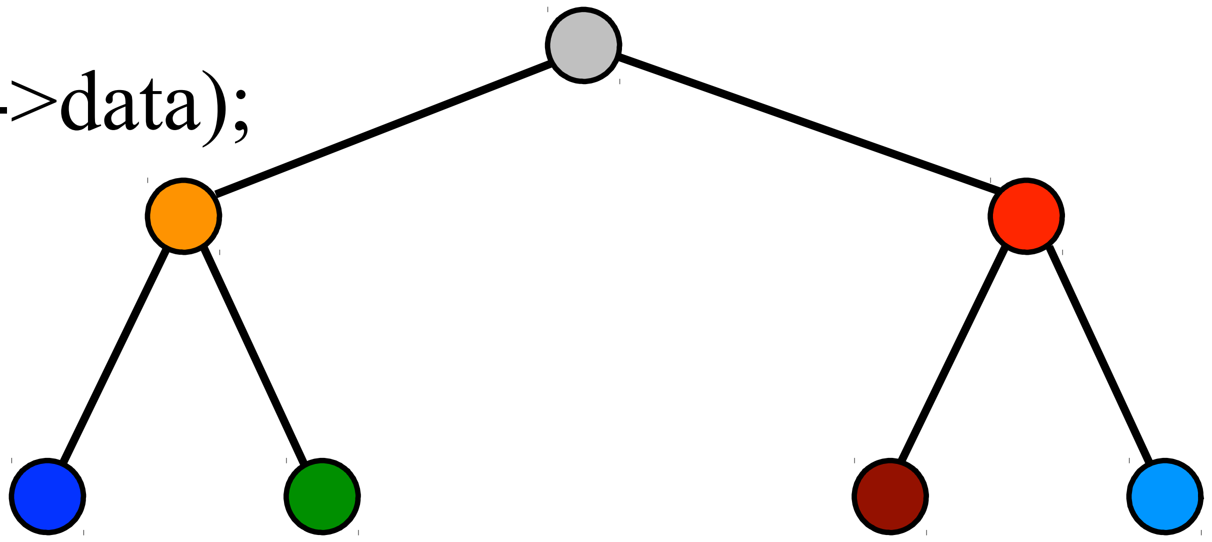




Postorder graph traversal in parallel — task wait

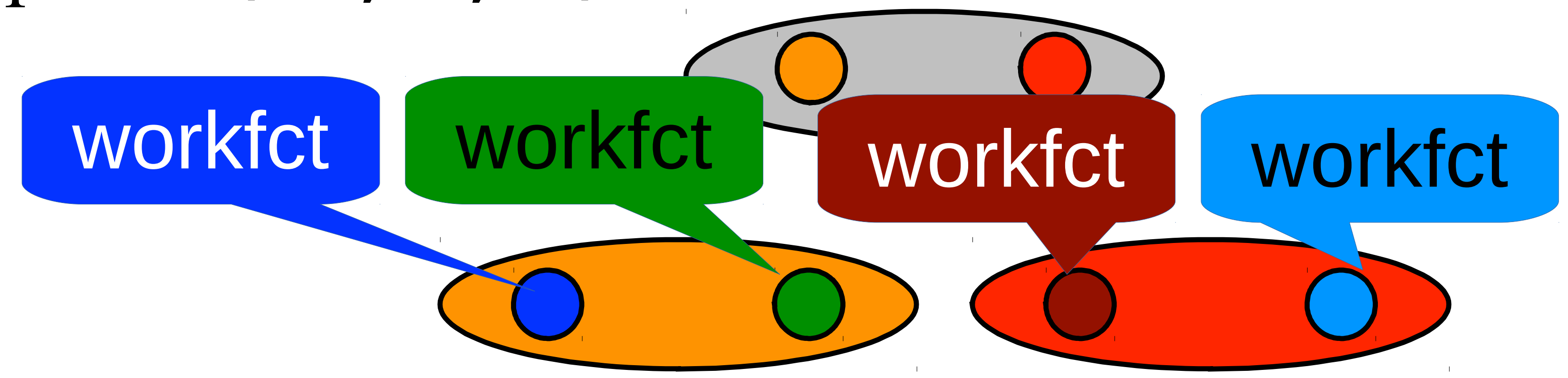
```
void postorder(node *p) { // p is ●
  if (p->left)
    #pragma omp task
          postorder(p->left);
  if (p->right)
    #pragma omp task
          postorder(p->right);
#pragma omp taskwait // wait for
descendants
  workfct(p->data);
}
```

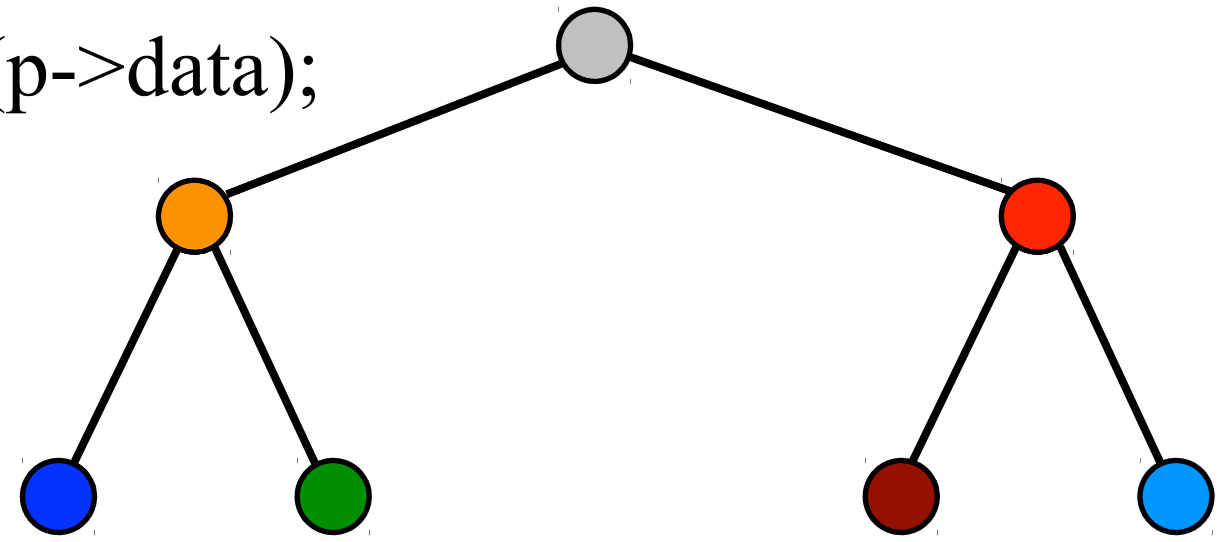




Postorder graph traversal in parallel — task wait

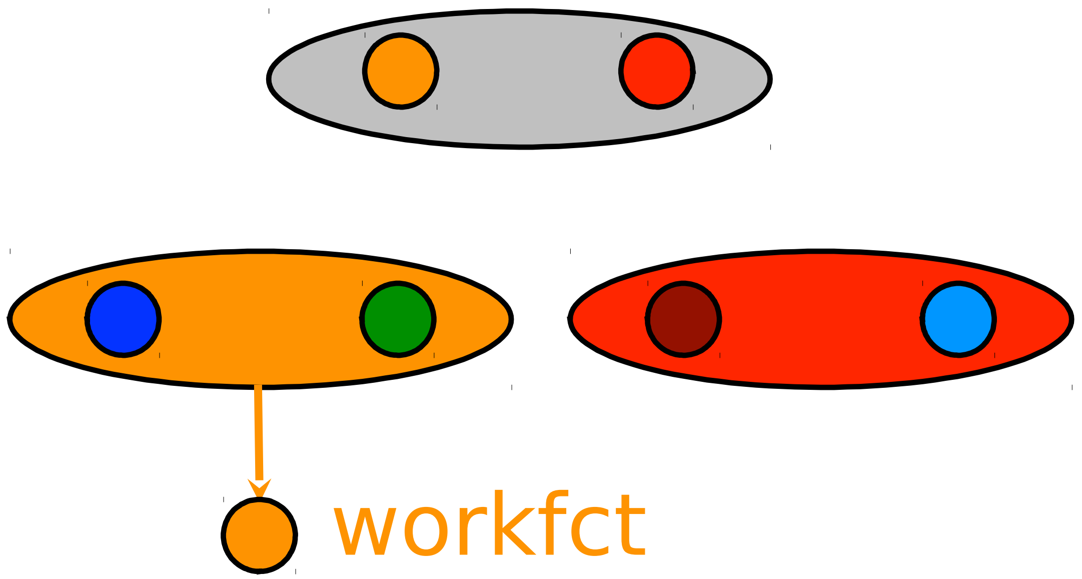
```
void postorder(node *p) { // p is ●, ●, ●, ●,
   if (p->left)
      #pragma omp task
            postorder(p->left);
   if (p->right)
      #pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for
descendants
   workfct(p->data);
}
```

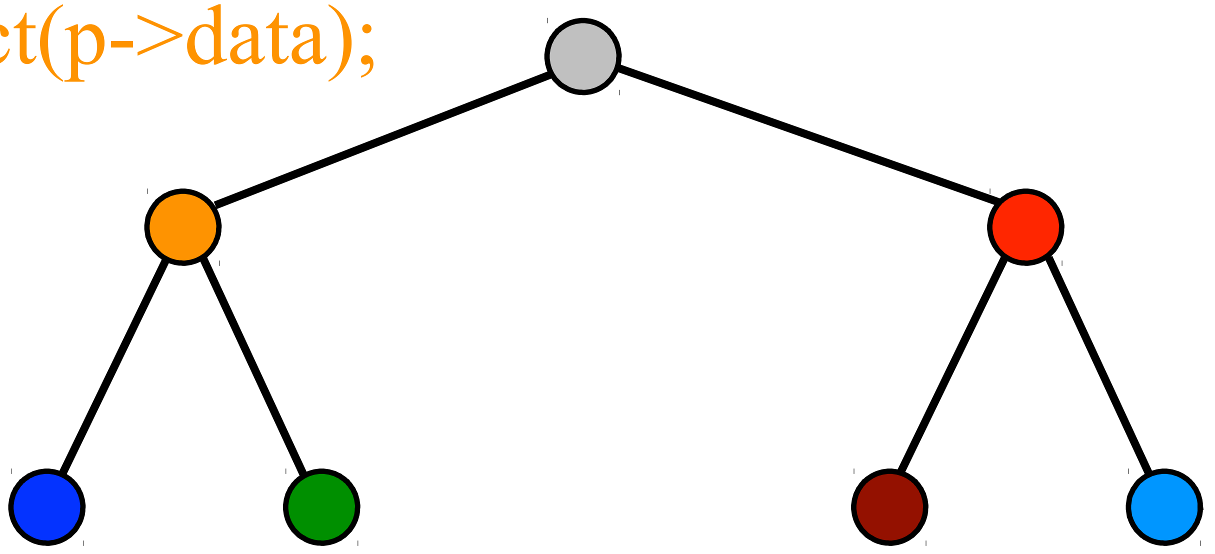




Postorder graph traversal in parallel — task wait

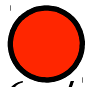
```
void postorder(node *p) { // p is ●
   if (p->left)
      #pragma omp task
            postorder(p->left);
   if (p->right)
      #pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for
descendants
   workfct(p->data);
}
```

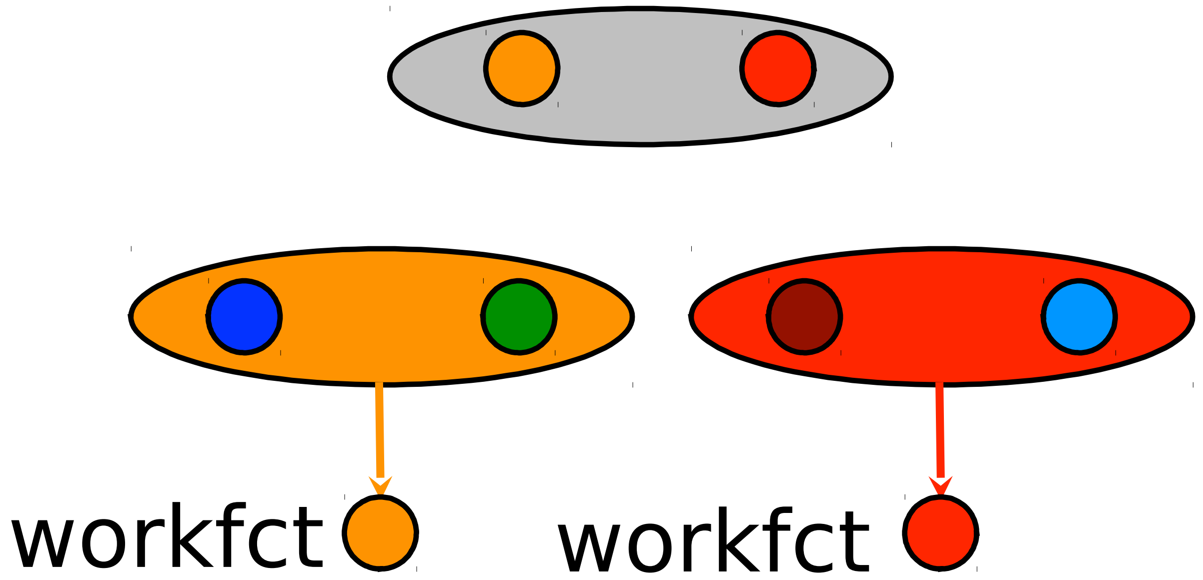


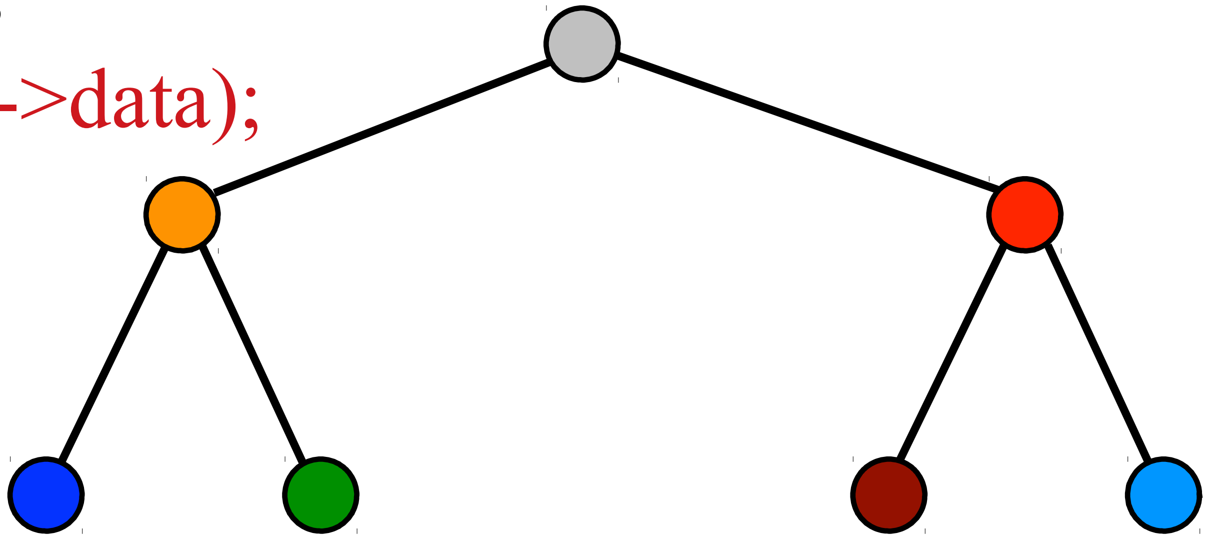


Postorder graph traversal in parallel — task wait

```
void postorder(node *p) { // p is
   if (p->left)
      #pragma omp task
            postorder(p->left);
   if (p->right)
      #pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for
descendants
   workfct(p->data);
}
```

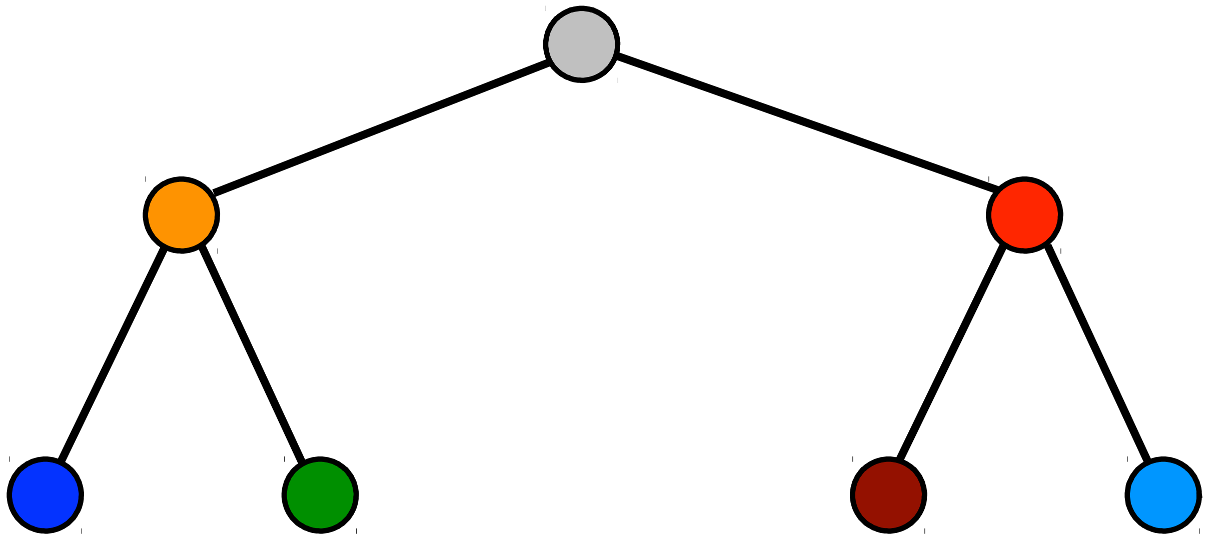


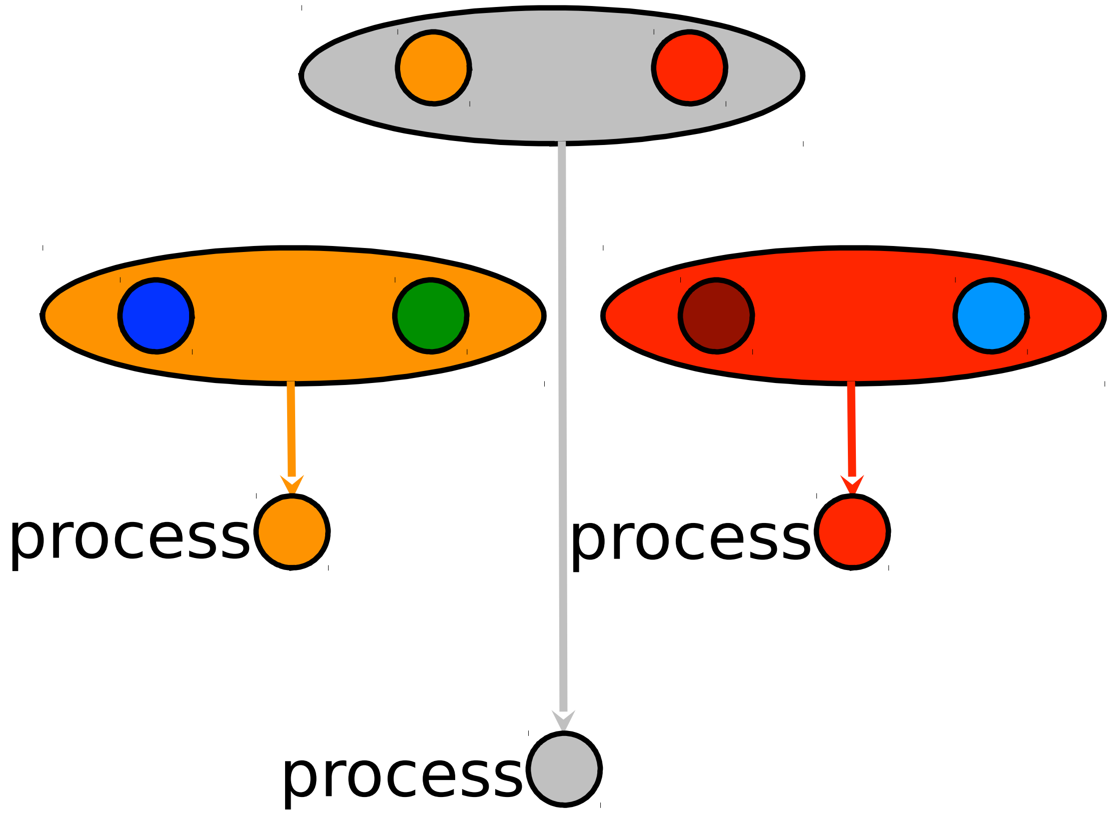




Postorder graph traversal in parallel — task wait

```
void postorder(node *p) {
   if (p->left)
      #pragma omp task
            postorder(p->left);
   if (p->right)
      #pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for descendants
   workfct(p->data);
}
```





Task scheduling points

- Certain constructs contain task scheduling points (*task constructs, taskwait constructs, taskyield* [*#pragma omp taskyield*] *constructs, barriers* (implicit and explicit), the end of a *tied* region)
- Threads at task scheduling points can suspend their task and begin executing another task in the task pool (*task switching)*
- At the completion of the task or at another task scheduling point it can resume executing the original task

Example: task switching

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
          process(item[i]);
}
```

- Many tasks rapidly generated -- *eventually more tasks than threads*
- Generated tasks will have to suspend until a thread can execute them
- With task switching, the executing thread can
  - execute an already generated task, draining the *task pool*
  - execute the encountered task (could be cache friendly)

Example: thread switching

```
#pragma omp single
{
  #pragma omp task untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task // tied
          process(item[i]);
}
```

The task generating other tasks is *untied*, the tasks executing *process( )* are tied.

- Eventually too many tasks are generated
- Task that is generating tasks is suspended and the task that is executed executes (for example) a long task
- Other threads execute all of the already generated tasks and begin starving for work
- With thread switching the task that generates tasks can be resumed by a *different* thread and generate tasks, ending starvation
- Programmer must specify this behavior with *untied*

sharing data

- Supported, but you have to be careful.
- Let *p* be a variable in a task T_1
- Let task T_1 spawn task T_2
- Let T_2 access *p shared* or *lastprivate*
- If there is no taskwait, T_1 can finish before T_2 does. When T_1 finishes, *p* no longer exists to be asscessed or copied back to.

Synchronization

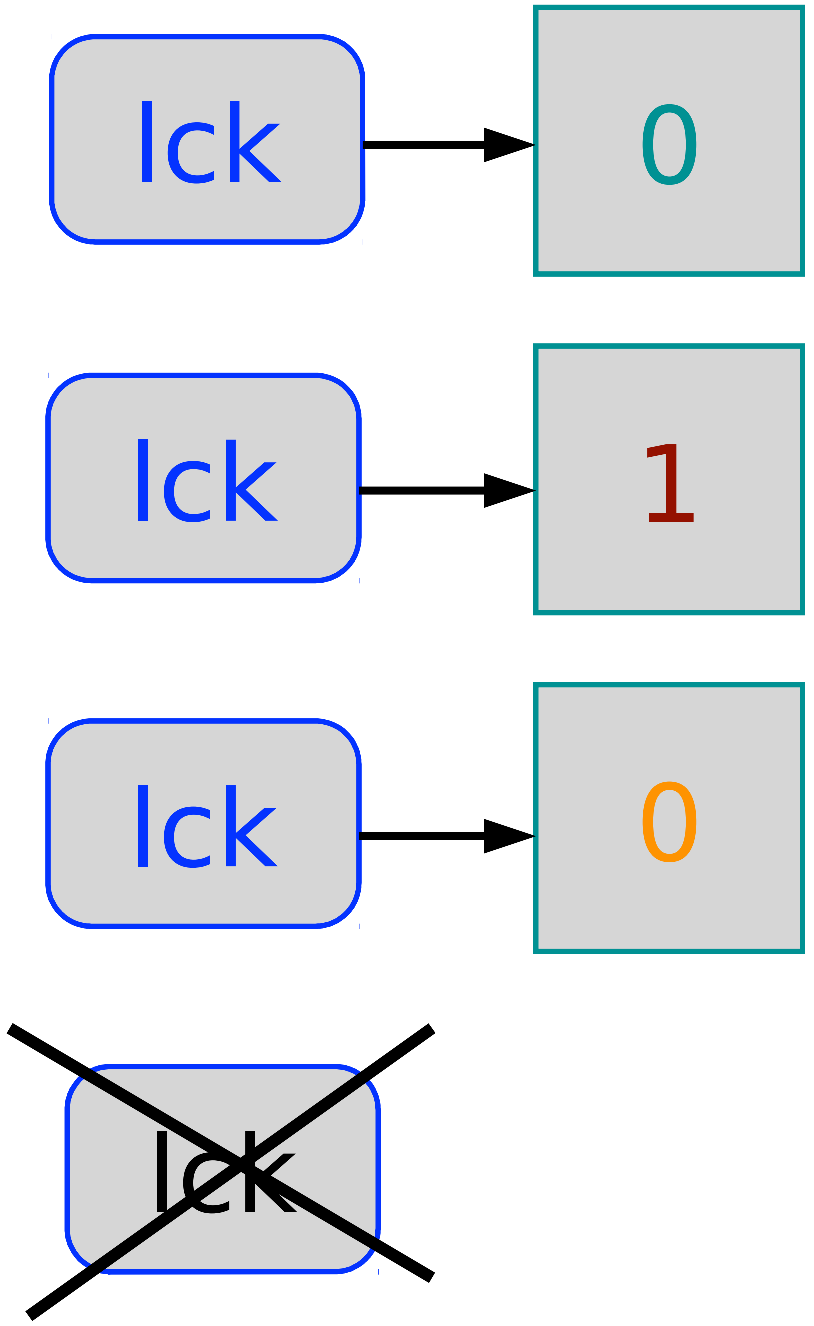
- Locks
- Nested locks

Simple locks

- A *simple lock* is available if it is not set
- Lock manipulation routines include:
  - omp_init_lock(...)
  - omp_set_lock(...)
  - omp_unset_lock(...)
  - omp_test_lock(...)
  - omp_destroy_lock

Simple lock example

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
  id = omp_get_thread_num();
  tmp = do_lots_of_work(id);
  omp_set_lock(&lck);
  printf(“%d %d”, id, tmp);
  omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```



Consider the code below . . .

```
void* items[100000000]; init(items);
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel for
{
  for (int i = 0; i < 100000000; i++) {
    omp_set_lock(&lck);
    update(items[i]);
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

```
void* items[100000000]; init(items);
#pragma omp parallel for
{
  for (int i = 0; i < 100000000; i++) {
    #pragma omp conflict
    update(items[i]);
}
```

Left and right code is pretty much the same and will essentially serialize the *for* loop.

Let’s try and do this with some actual parallelism

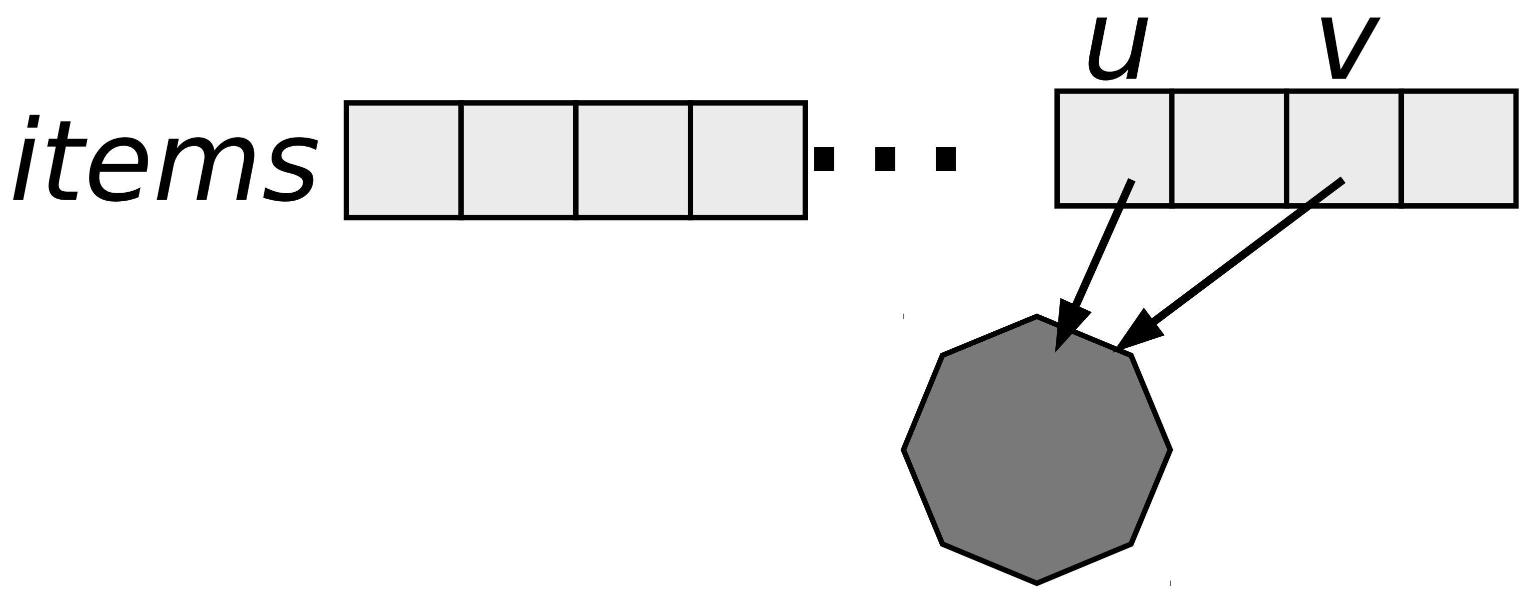
```
void* items[100000000]; init(items); // items[i] and items[j] may point to
                                     // the same thing
omp_lock_t lck[100000000];
for (int i = 0; i < 100000000; i++)
  omp_init_lock(&(lck[i]));
#pragma omp parallel for
{
  for (int i = 0; i < 100000000; i++) {
    omp_set_lock(&(lck[i]));
    update(items[i]);
    omp_unset_lock(&(lck[i]));
}
for (int i = 0; i < 100000000; i++)
  omp_destroy_lock(&(lck[i]));
```

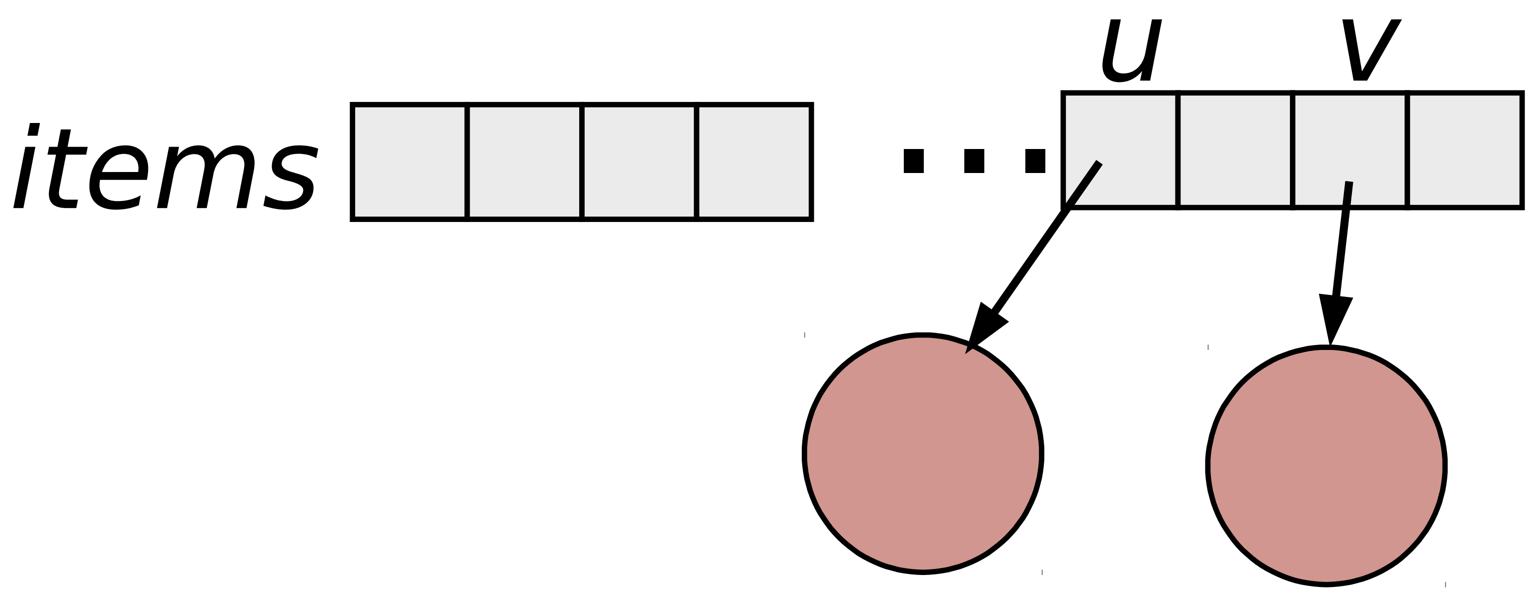
This doesn’t work, why?

Hint: what is being changed by update and what does the set lock correspond to?

Why it is wrong

- items[u] and items[v] point to the same storage/object
- two different locks are acquired/set by
  `omp_set_lock(&(lck[u]));`
  `omp_set_lock(&(lck[v]));`
- Locks are not providing exclusive access to the object
- Also, there are implementation limits on the number of locks

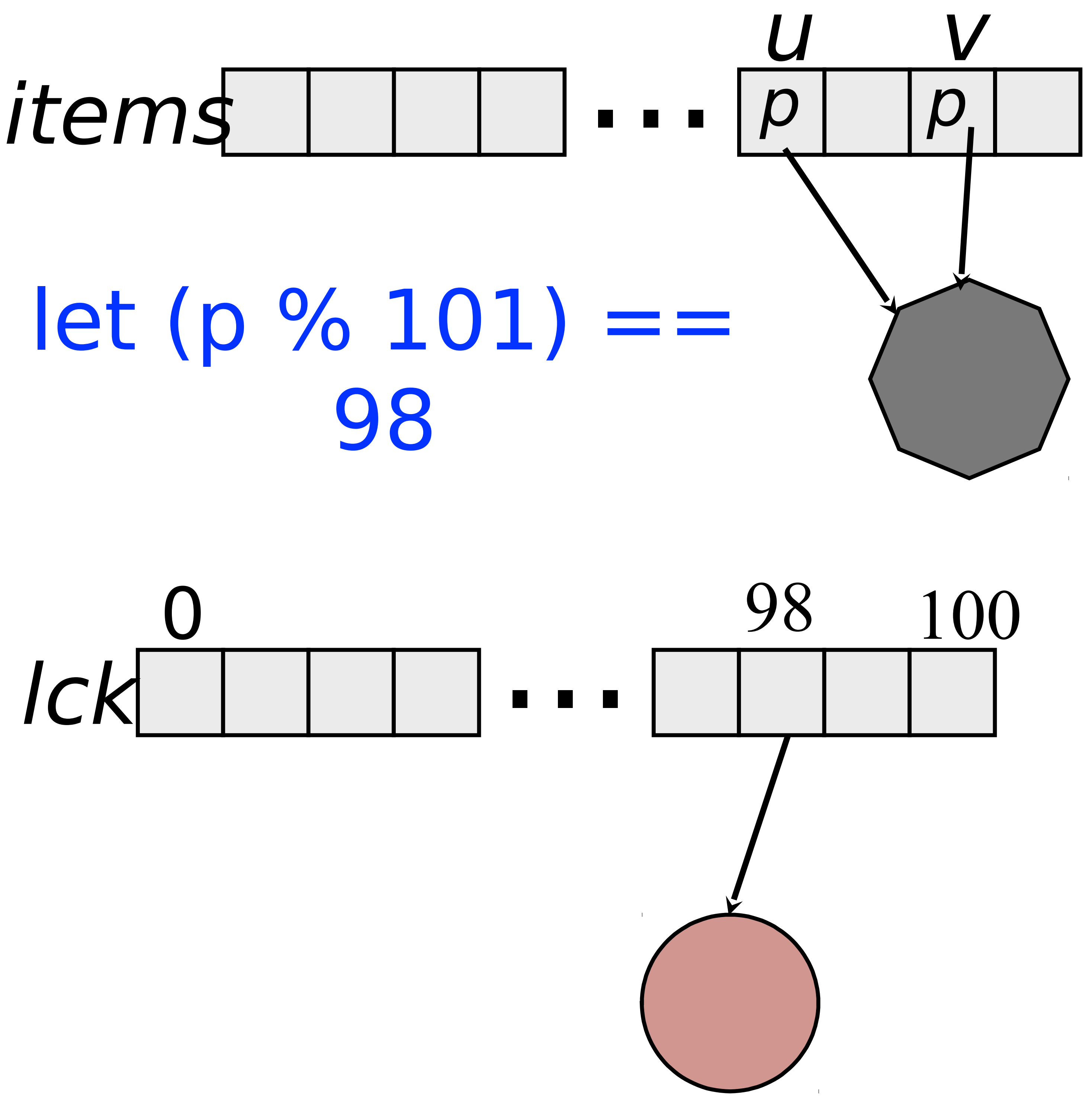




The right (or at least better) way to do this

```
void* items[100000000]; init(items); // items[i] and items[j] may point
to
                                       // the same thing
omp_lock_t lck[101];
for (int i = 0; i < 101; i++)
  omp_init_lock(&(lck[i]));
#pragma omp parallel for private(tmp)
{
  for (int i = 0; i < 100000000; i++) {
    int tmp = (((int) items[i]) % 101));
    omp_set_lock(&(lck[tmp]));
    update(items[i]);
    omp_unset_lock(&(lck[tmp]));
}
for (int i = 0; i < 101; i++)
  omp_destroy_lock(&(lck[i]));
```

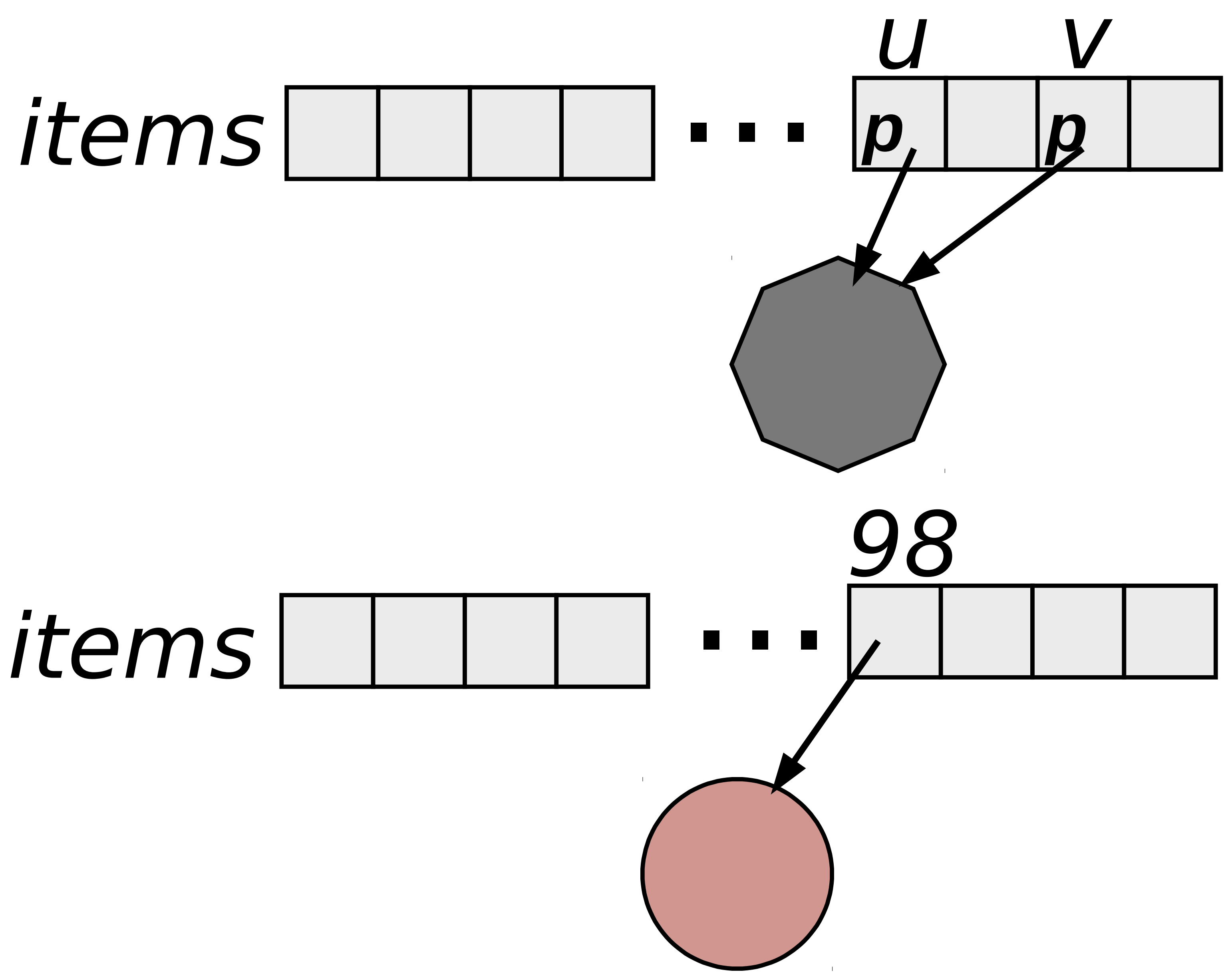
Why this works



- If pointers are evenly distributed then few collisions on << 101 threads, little serialization
- Balance the number of locks to give an acceptable chance of collision on a lock

Why this works

Let p%101 = 98

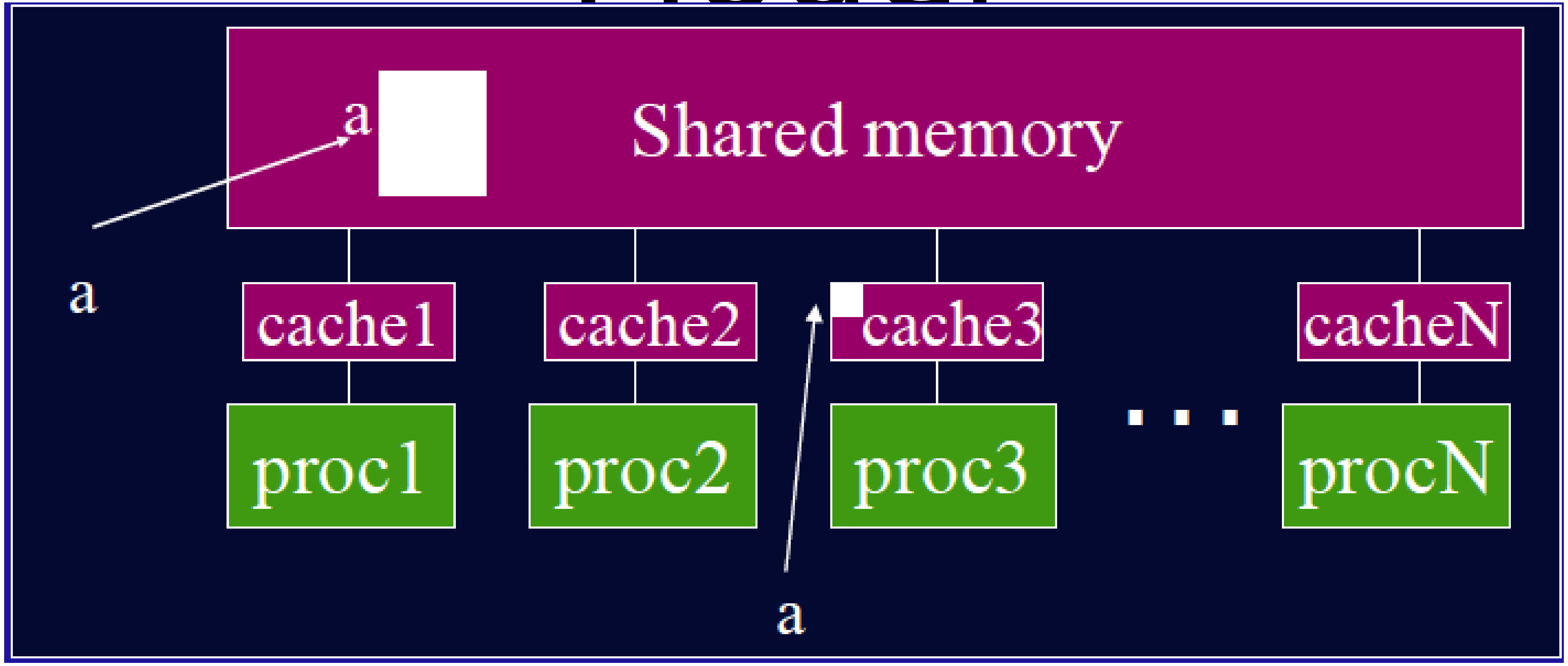


- If pointers are evenly distributed then few collisions on << 101 threads, little serialization
- Balance the number of locks to give an acceptable chance of collision on a lock

Nested locks

- A *nested lock* is available if it is not set or it is set by the same thread attempting to acquire it.
- Lock manipulation routines include:
  - omp_init_nest_lock(...)
  - omp_set_ nest_ lock(...)
  - omp_unset_ nest_ lock(...)
  - omp_test_ nest_ lock(...)
  - omp_destroy_ nest_lock

OpenMP Memory Model

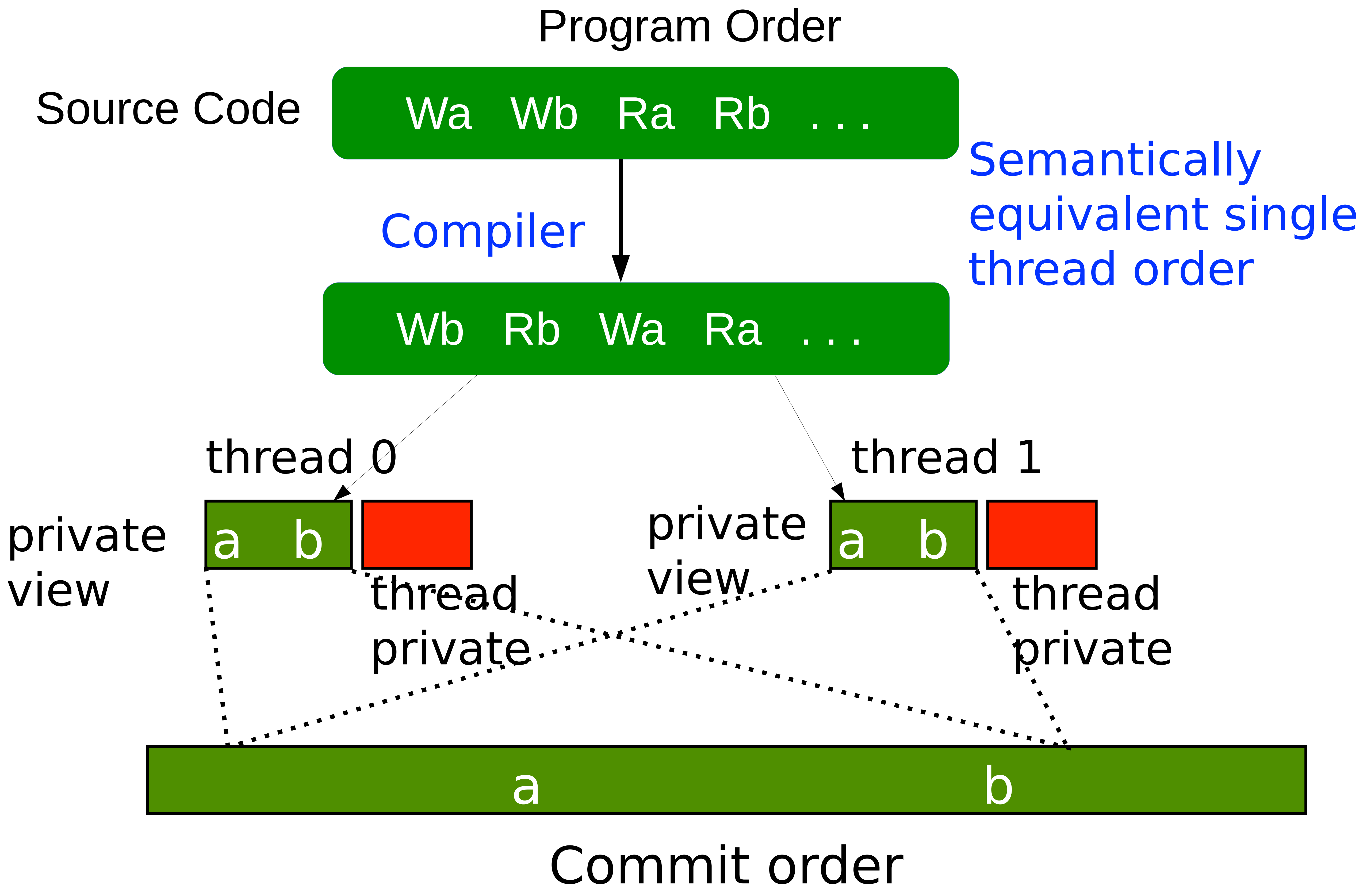


Two issues, coherence and consistency.
Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
Consistency: Orderings of accesses to different addresses by multiple threads.

Memory models

- Memory models define the interactions of loads and stores (reads and writes) *in different threads*
- HW dependences (*hazards)* are used to deal with reads and writes within a thread *to the same memory location* and are not generally thought of as part of the memory model.
  - Stated differently, regardless of of the memory model, reads/writes, writes/writes and writes/reads within a thread to the same memory location will be in-order

OpenMP Memory Model Basics



Sequential Consistency

- An operation is sequentially consistent (SC) if the operation is in the same order in the program order, code order and commit order.
- An execution is SC if all operations *appear* to be SC
- A *consistency model* where all operations are SC is *strict*
- A consistency model where some of these orders can be violated is *relaxed.*
- Most languages and processors have relaxed orders

Reordering Accesses

- Compiler reorders program order to code order
  - Reordering happens because of the compiler doing optimizations. In practice, compilers *will maintain SC if the program is well-synchronized*, for reasons we will see soon.
- Hardware reorders *code order* to *commit order*
  - Reordering happens because of out-of-order execution. Hardware *will maintain SC if the code order is SC and the program is well synchronized*.
- The private view of memory can differ from shared memory
- Consistency models are based on orderings of Reads (R), Writes (W) and Synchronizations (S) *within a thread*

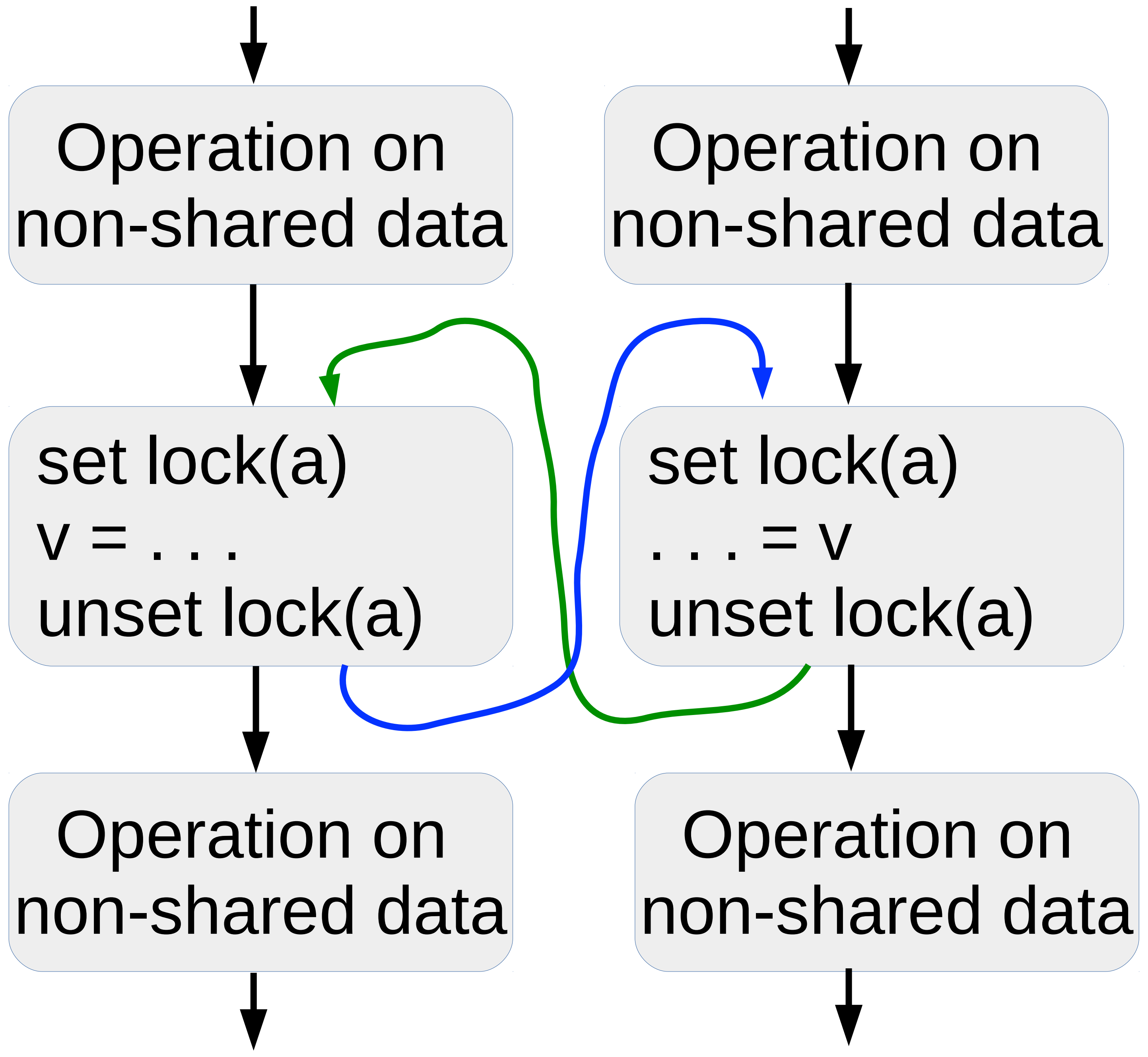
R→R, W→W, R→W, W→R, R→S, S→S, W→S

OpenMP’s consistency model

- Weak consistency
- S ops (synchronization operations) must be executed in sequential order
  - Within a thread cannot reorder S with respect to W or S with respect R (cannot move past a read or write)
  - Guarantees S→W, S→R, R→S, W→S, S→S
  - R→R, W→W, R→W missing. Obviously, if writes or read/writes to the same location they are ordered (dependences/hazards enforced) *If read or write not to same memory location, can be moved around with respect to one another*

What is a race?

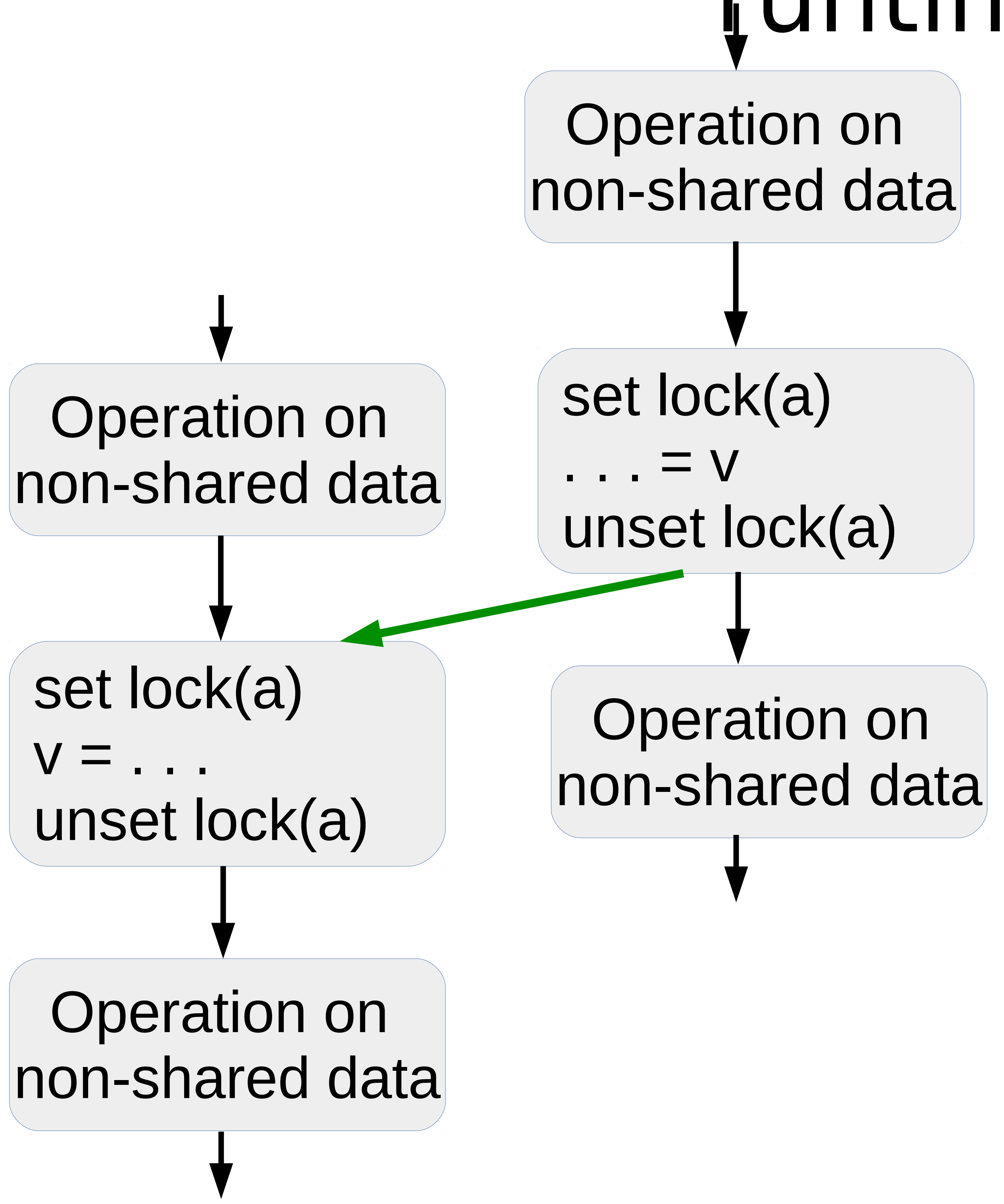
- Execute a parallel program
- If a there is a read or write to some *v* in a thread, and a write to it in another thread, and no enforced ordering *at runtime* between the two, there is a race.
- *Orderings come from synchronization*



Operation on non-shared data
Operation on non-shared data
set lock(a)
v = . . .
unset lock(a)
set lock(a)
. . . = v
unset lock(a)
Operation on non-shared data
Operation on non-shared data

either blue or green order *must* exist at runtime

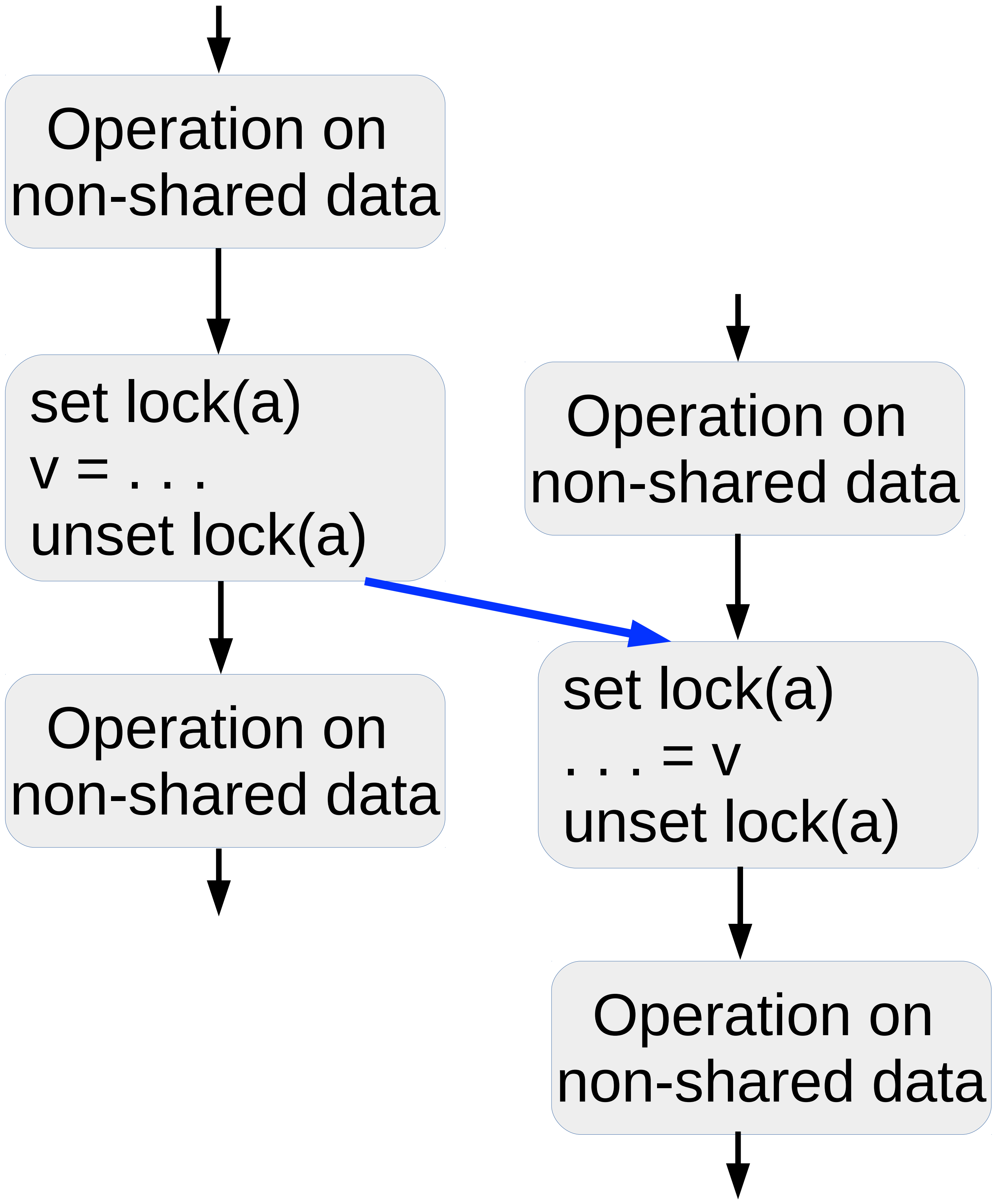
Green order occurs at runtime



Operation on non-shared data
set lock(a)
. . . = v
unset lock(a)
Operation on non-shared data
Operation on non-shared data
set lock(a)
v = . . .
unset lock(a)
Operation on non-shared data

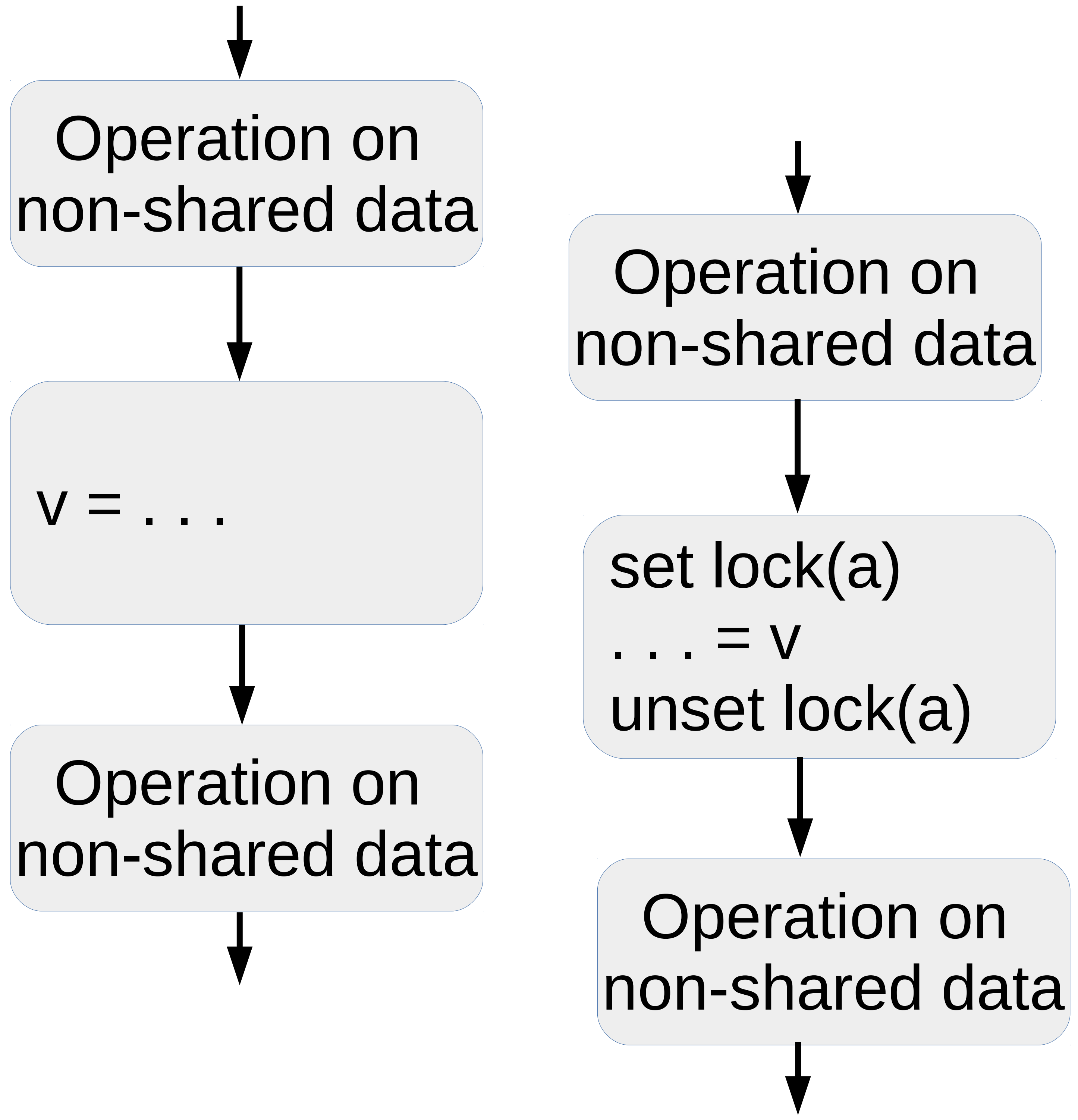
write to v must occur after the read -- cannot be overlapping

Blue order occurs at runtime


Operation on non-shared data
set lock(a)
v = . . .
unset lock(a)
Operation on non-shared data
Operation on non-shared data
set lock(a)
. . . = v
unset lock(a)
Operation on non-shared data

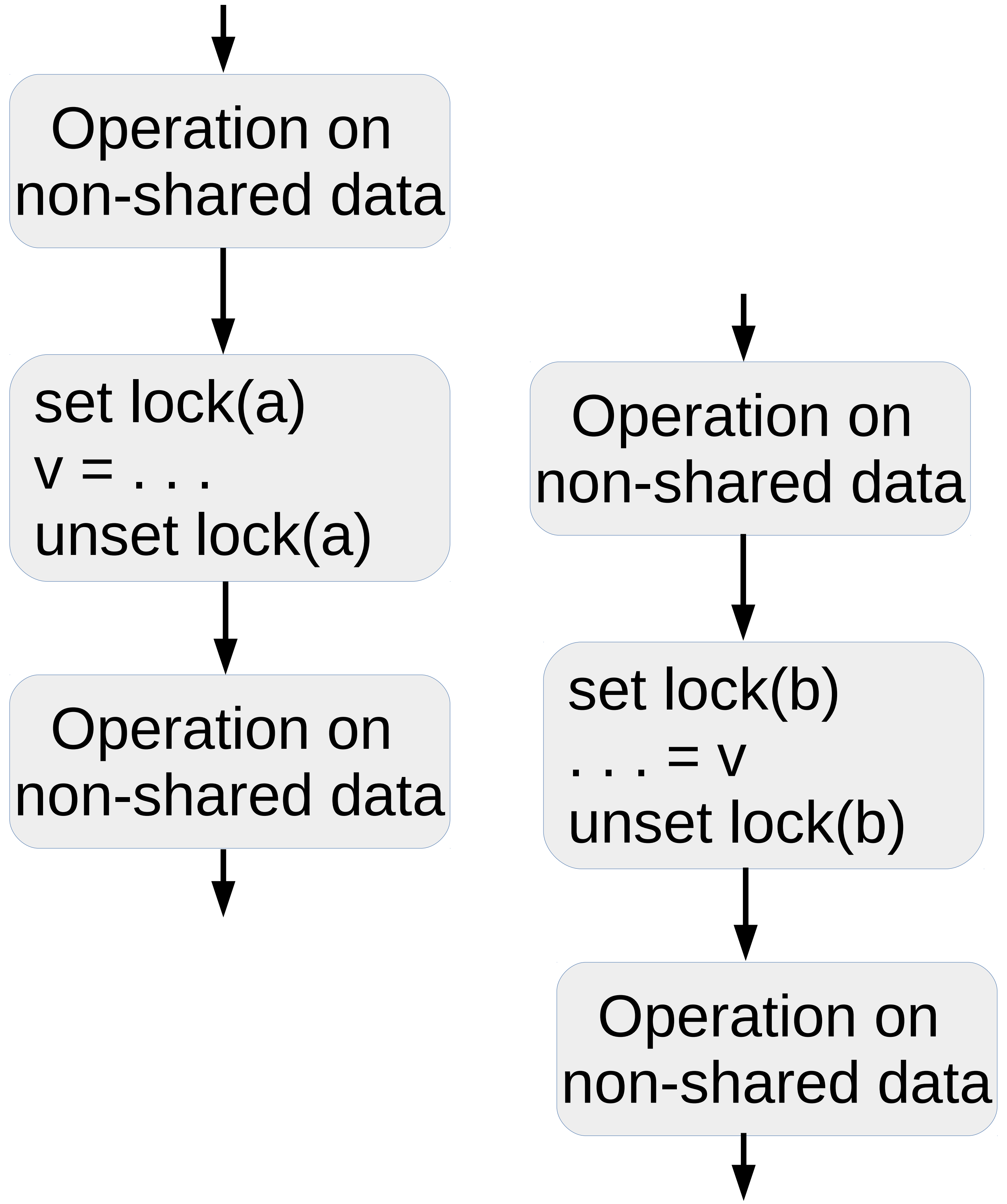
Read and write of V cannot overlap since write must occur before read

A race exists – both accesses are not enforce by a lock


Operation on non-shared data
v = . . .
Operation on non-shared data
Operation on non-shared data
set lock(a)
. . . = v
unset lock(a)
Operation on non-shared data

A race exists – there is no ordered path from the read in one thread to the write in the other, or *vice versa*

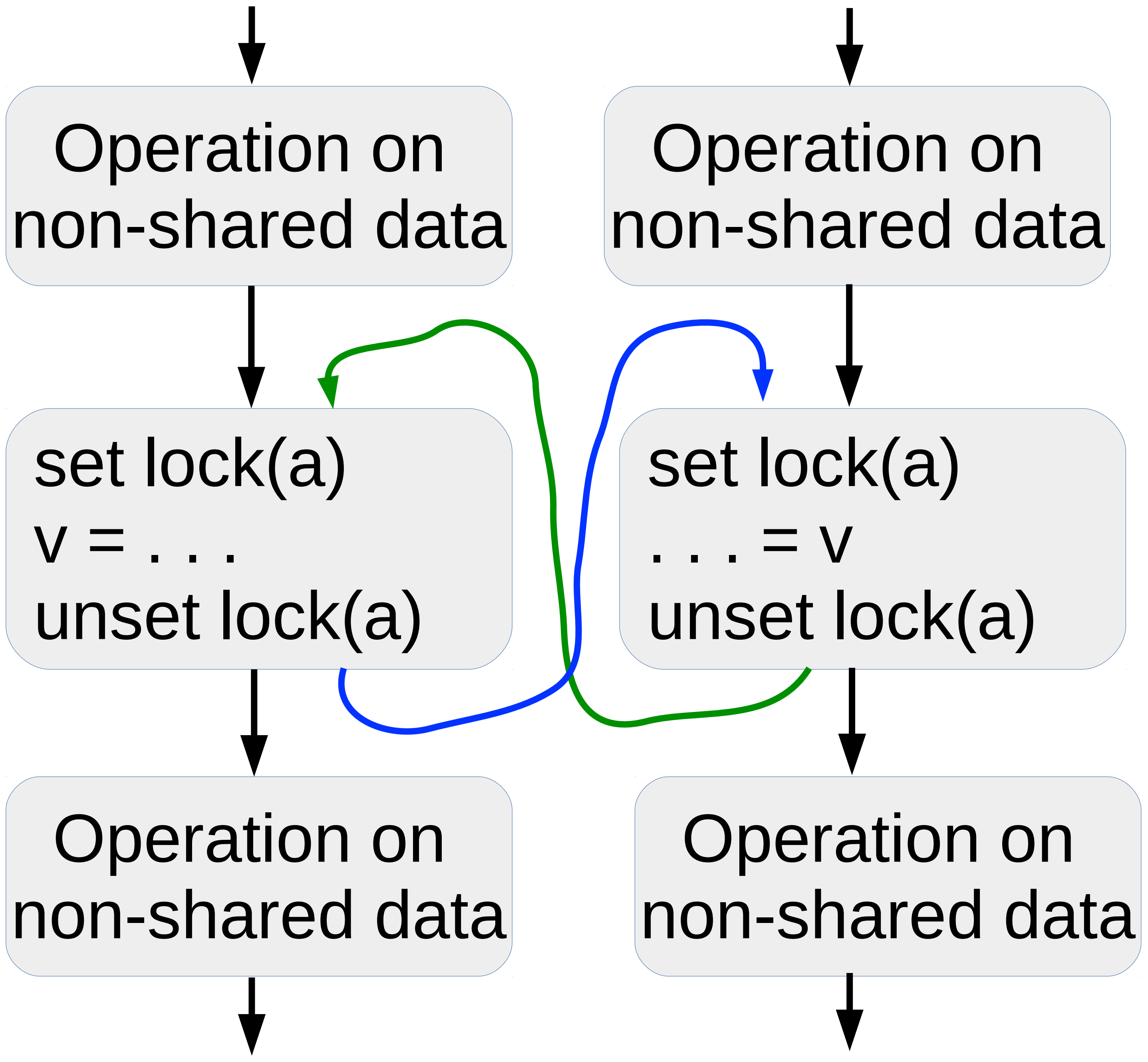
A race exists – the read and write of v are not guarded by the same lock


Operation on non-shared data
set lock(a)
v = . . .
unset lock(a)
Operation on non-shared data
Operation on non-shared data
set lock(b)
. . . = v
unset lock(b)
Operation on non-shared data

For an order to exist between v= and =v it must be that the *fence* in the unset_lock( ) forces any new value of v out before the unset_lock completes

The fence will not complete until the value to memory is committed

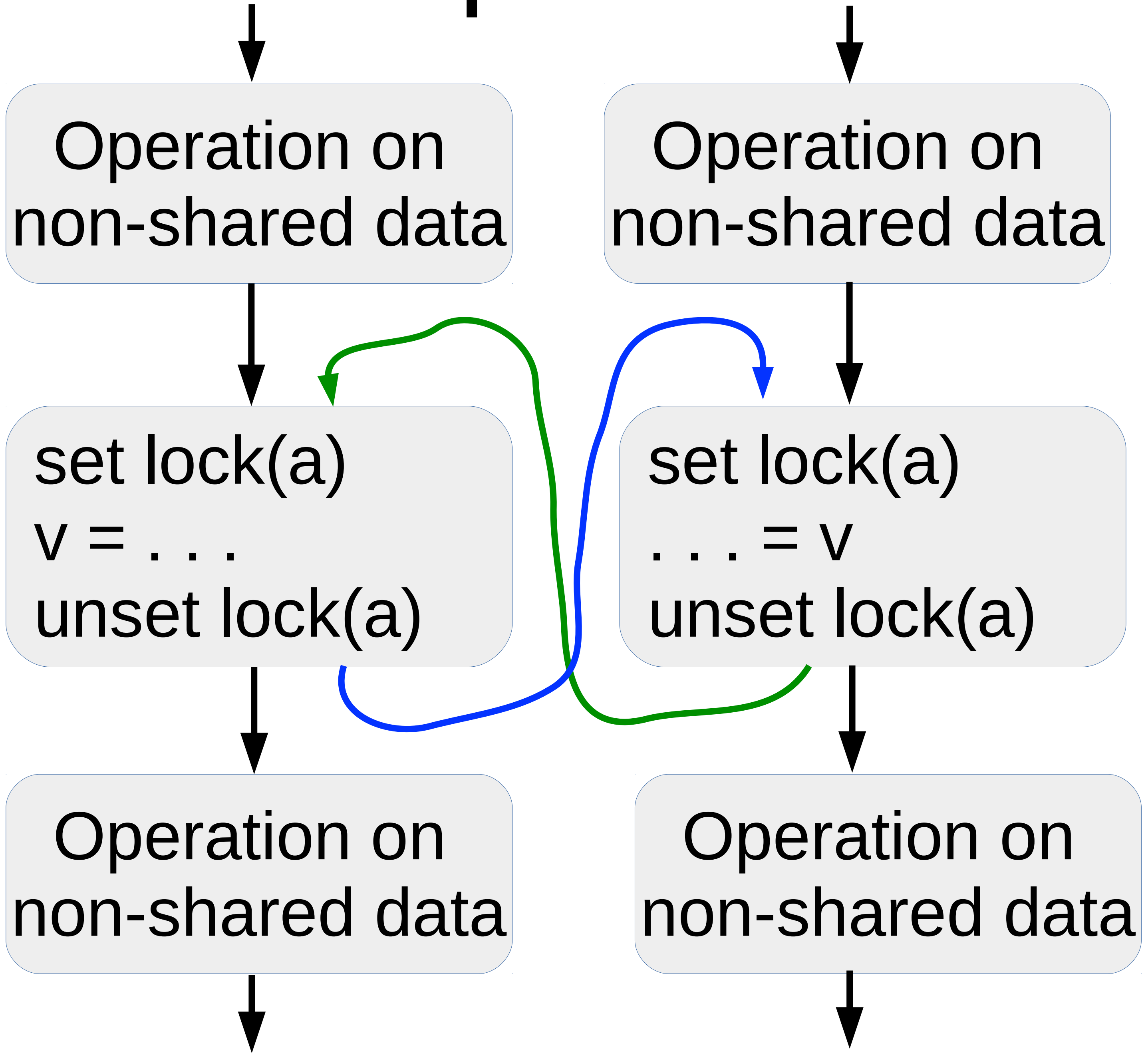
The value to memory will not be committed before any stale values of v are invalidated


Operation on non-shared data
Operation on non-shared data
set lock(a)
v = . . .
unset lock(a)
set lock(a)
. . . = v
unset lock(a)
Operation on non-shared data
Operation on non-shared data

What about IBM's Power processors?

Some Power fence's (called *sync* instructions) can complete before the value is committed to memory. I.e., value may be committed to shared cache or local memory.

This makes for harder low-level programming but may make the machine faster (*sync*'s execute faster)


Operation on non-shared data
Operation on non-shared data
set lock(a)
v = . . .
unset lock(a)
set lock(a)
. . . = v
unset lock(a)
Operation on non-shared data
Operation on non-shared data

The OpenMP standard requires that OpenMP fences on Power processors wait until new value visible to all and old values invalidated

Remember that local view and shared memory may not be the same

- *flush* forces a consistent view between the local and shared memory by executing a fence
- *flush( )* flushes all thread visible variables
- *flush(list)* flushes all variables in *list*
- A *flush* guarantees that
  - all read and writes ops that read or write data in *list* and that are before the *flush( )* will complete before the flush completes
  - all read and writes ops that read or write data in *list* and that are after the *flush( )* will not start before the flush completes
  - flushes with overlapping lists (flush sets) cannot be re-ordered with respect to one another *in the same thread*
- Locks always execute a flush, as do barriers.

Flush Example

- The flush ensures that other threads can see *A* after the flush executes
- Serves the function of a *fence* in hardware API's

```
double A;
A = compute();
flush(A); // flush to memory to
          // make sure other
          // threads can pick up
          // the right value
```

I can't think of a good use of it in a non-racy program since unlock essentially does a flush

Compilers and flushes

- Compilers routinely reorder instructions
- Compilers cannot move a read or write past a barrier or a flush whose *flush set* contains the read or written variable
- Keeping track of what is consistent can be confusing for programmers, especially if *flush(list) is used*
- flushes *do not* synchronize -- they make local and shared memory consistent for the thread executing flush

Runtime library calls

- omp_set_dynamic(true|false) (default is true)
- omp_get_dynamic( ) (test function)
- omp_num_procs( )
- omp_in_parallel( )
- omp_get_max_threads( )
- omp_thread_limit
- double omp_get_wtime( )
- double omp_get_wtick( );

Nested parallelism

- You can nest parallelism constructs
- Calling *omp_set_num_threads( )* within a parallel construct sets the number of threads available to the *next* level of parallelism
- Can get info about execution environment:

omp_get_active_level() // level of
// parallelism nesting

omp_get_ancestor(level) // thread ID of an
// ancestor

omp_get_teamsize(level) // number of
// threads executing an ancestor

Functions to control the level of allowed nested parallelism

- Can set maximum active levels of parallelism

OMP_MAX_ACTIVE_LEVELS (environment variable)
omp_set_max_active_levels()
omp_get_max_active_levels

Loops

```
$omp parallel for schedule(static) nowait
for (i=0; i < n; i++) {
   a(i) = ....
}
$omp parallel for schedule(static)
for (j=0; j < n; j++) {
   ... = a(j)
}
```

Guarantees iterations for both loops to execute on the same threads

Loops

```
$omp parallel for collapse(2)
for (i=0; i < n; i++) {
   for (j=0; j < n; j++) {
      .....
   }
}
```

forms a single parallel loop with n*n iterations

Loops (cont.)

- Schedule runtime (*schedule(runtime)*) made more useful. Can set at runtime rather than just reading from the environment

omp_set_schedule()
omp_get_schedule()

```
omp_set_schedule(omp_sched_static, 5);
```

AUTO schedule now supported --
runtime picks a schedule

C++ Random access iterators
can be used as control variables
in parallel loops

Portability

- Environment variables to control stack size added: *omp_stacksize*
- Added environment variable to specify how to handle idle threads: *omp_wait_policy*

ACTIVE: keep threads alive at barriers/locks

PASSIVE: try to release threads to the processor (i.e., don’t use CPU cycles

  - If not set, active for a while at barrier, then passive.

- Can specify maximum number of threads to use

OMP_THREAD_LIMIT

omp_get_thread_limit( )