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

- 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 threads

- 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)
- 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* generated in the current task, not to tasks generated by those tasks *T*

Example: parallel pointer chasing

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

process is an ordinary user function.

Example: parallel pointer chasing

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

Example: parallel pointer chasing

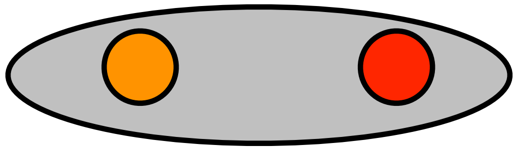
```
void postorder(node *p) {
   if (p->left)
      #pragma omp task
         postorder(p->left);
   if (p->right)
      #pragma omp task
         postorder(p->right);
#pragma omp taskwait // wait for descendants
   process(p->data);
}
```

Parent task suspended until child tasks finish

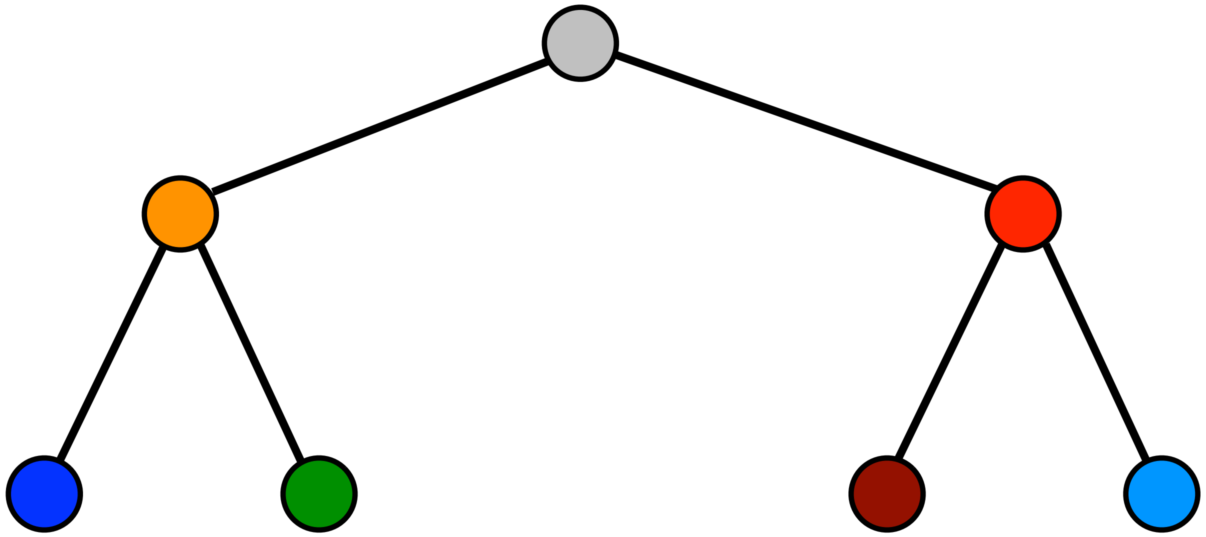
This is a task scheduling point

Example: parallel pointer chasing

```
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
  process(p->data);
}
```

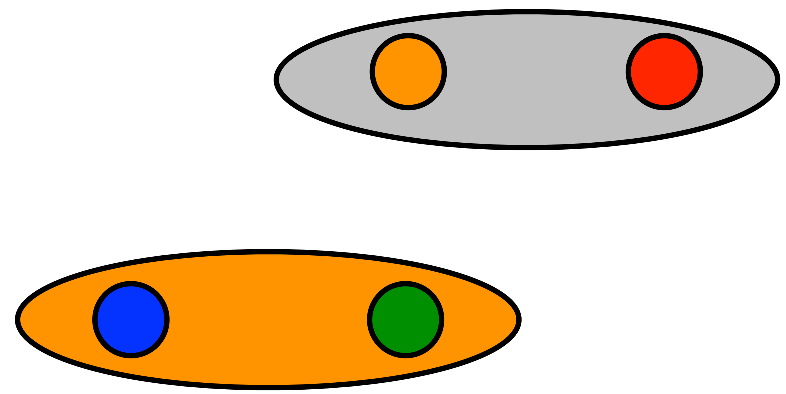


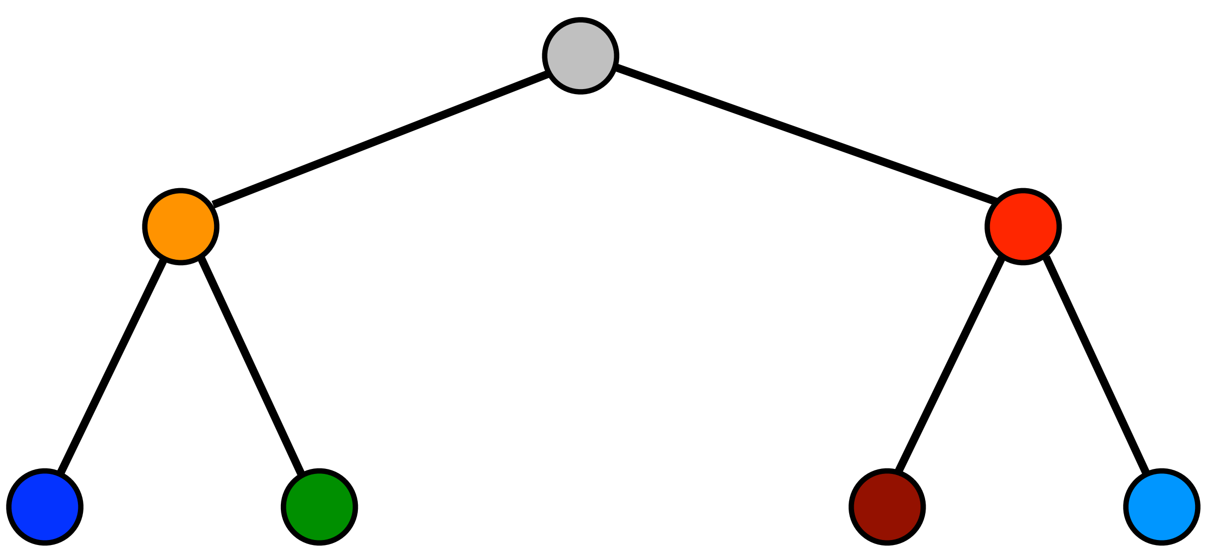
Code is within an omp parallel section



Example: parallel pointer chasing

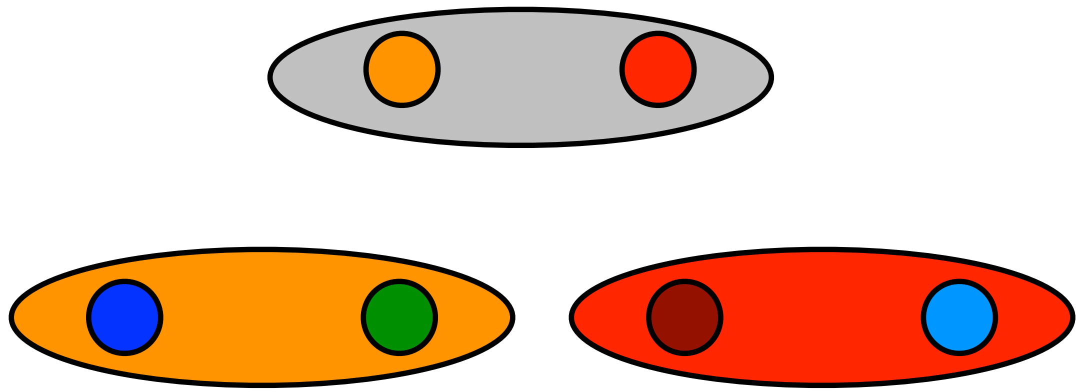
```
void postorder(node *p) { // p is
  if (p->left)
    #pragma omp task
        postorder(p->left);
  if (p->right)
    #pragma omp task
        postorder(p->right);
#pragma omp taskwait // wait for descendants
  process(p->data);
}
```

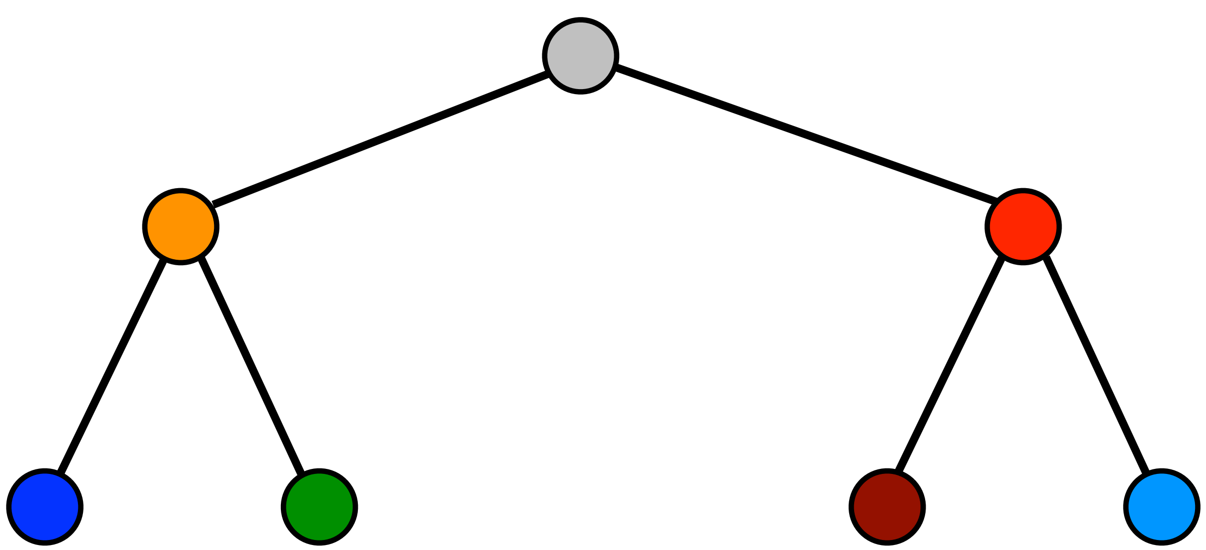




Example: parallel pointer chasing

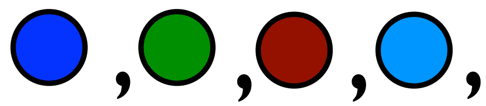
```
void postorder(node *p) { // p is
  if (p->left)
    #pragma omp task
          postorder(p->left);
  if (p->right)
    #pragma omp task
          postorder(p->right);
#pragma omp taskwait // wait for descendants
  process(p->data);
}
```

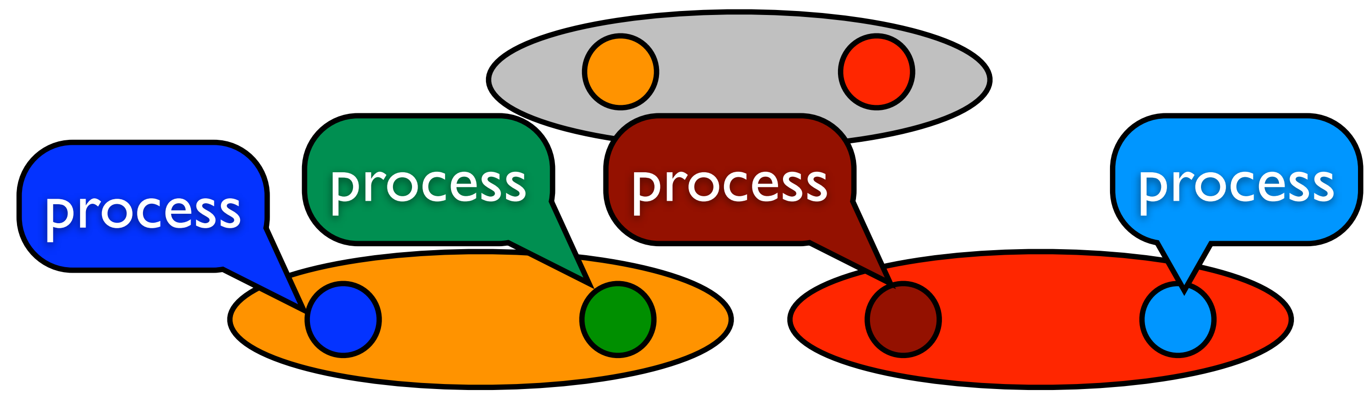


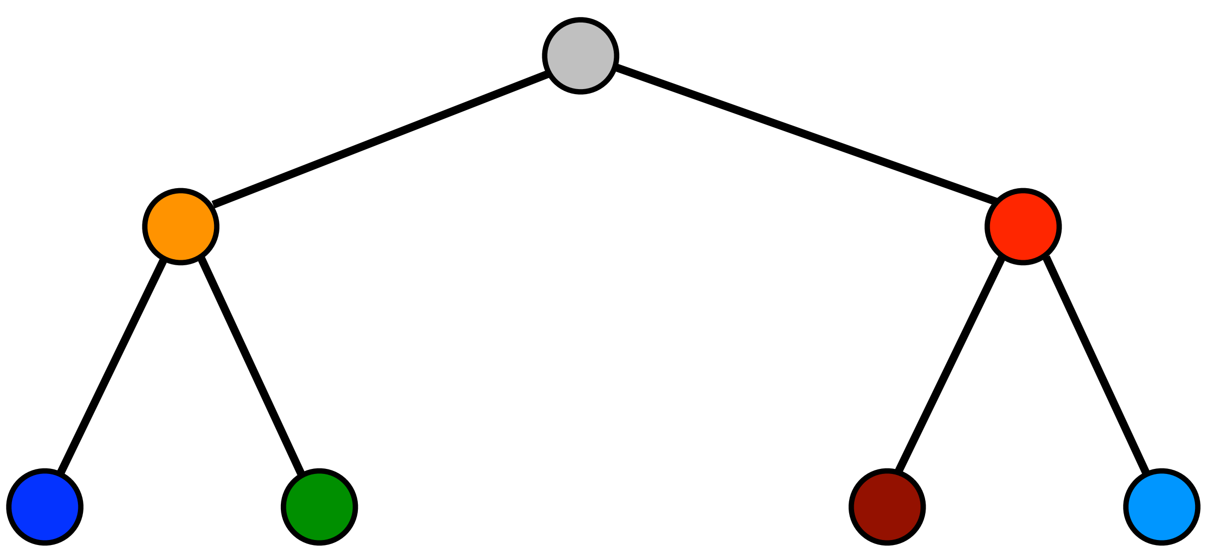


Example: parallel pointer chasing

```
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
  process(p->data);
}
```

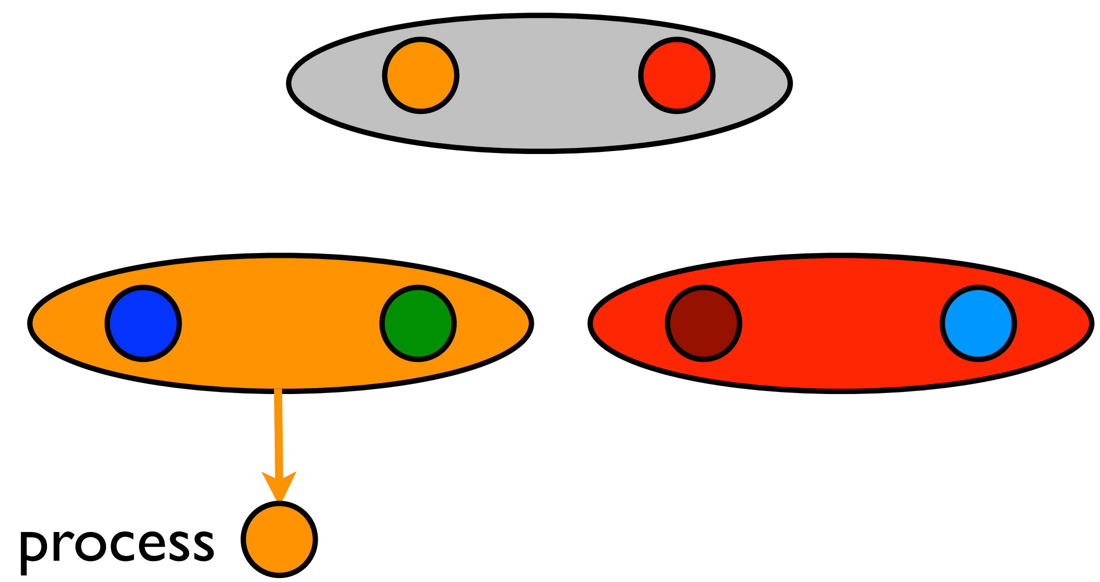


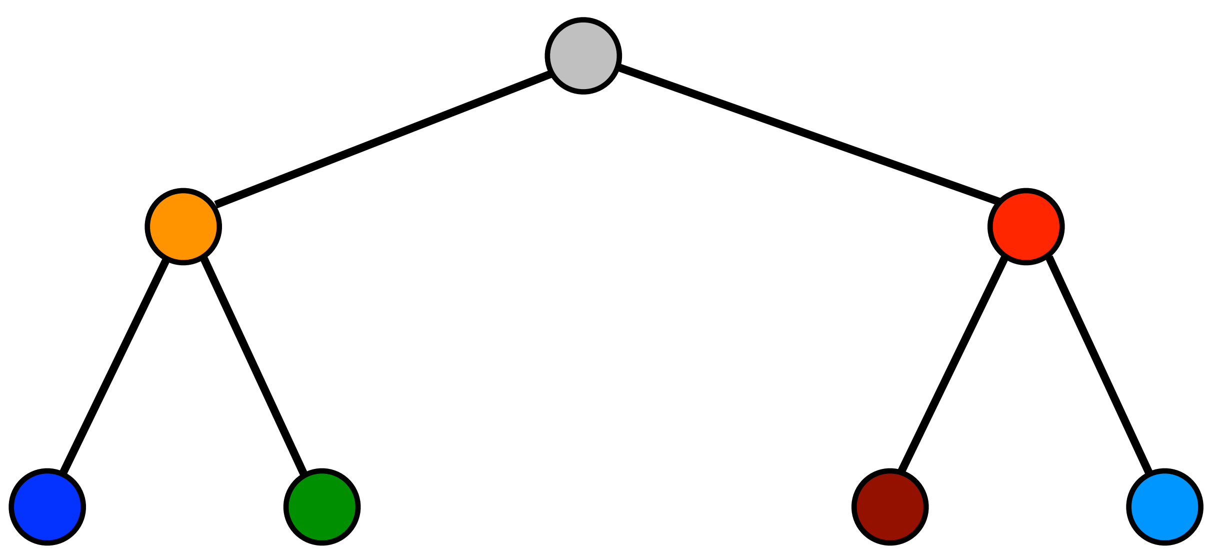




Example: parallel pointer chasing

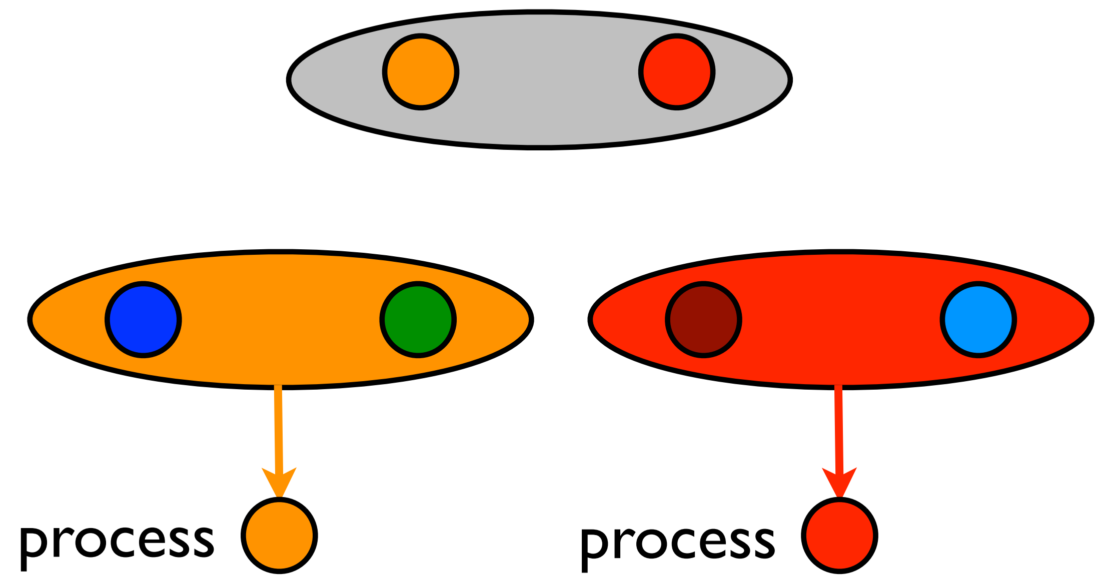
```
void postorder(node *p) { // p is
  if (p->left)
    #pragma omp task
          postorder(p->left);
  if (p->right)
    #pragma omp task
          postorder(p->right);
#pragma omp taskwait // wait for descendants
  process(p->data);
}
```

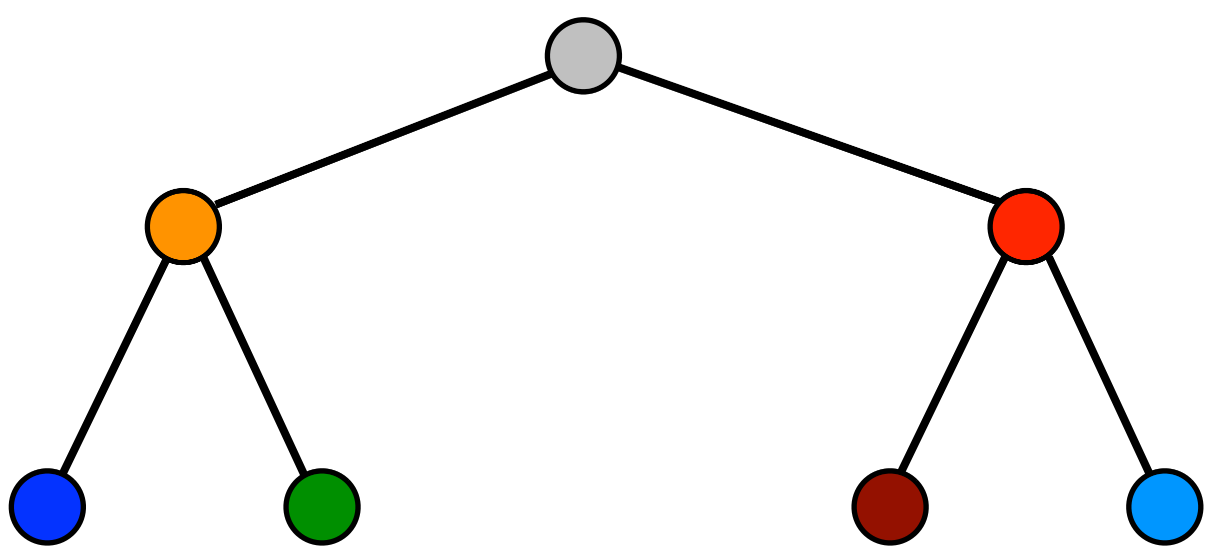




Example: parallel pointer chasing

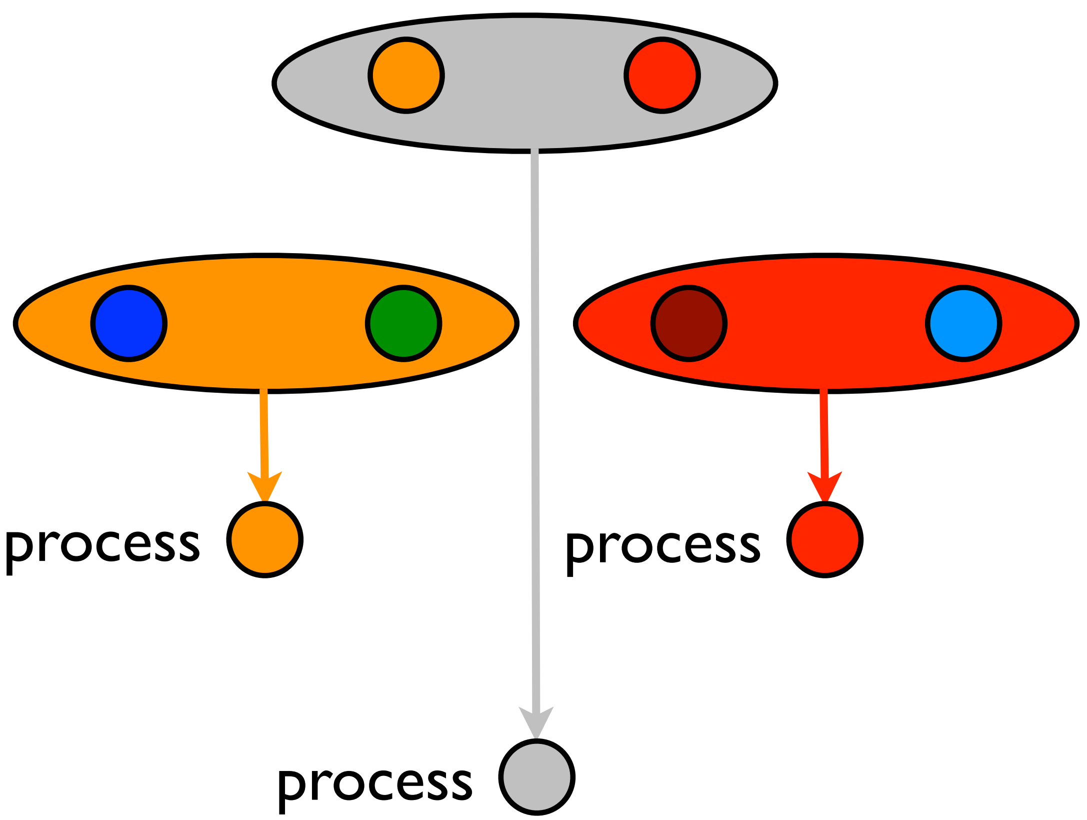
```
void postorder(node *p) { // p is
  if (p->left)
    #pragma omp task
          postorder(p->left);
  if (p->right)
    #pragma omp task
          postorder(p->right);
#pragma omp taskwait // wait for descendants
  process(p->data);
}
```

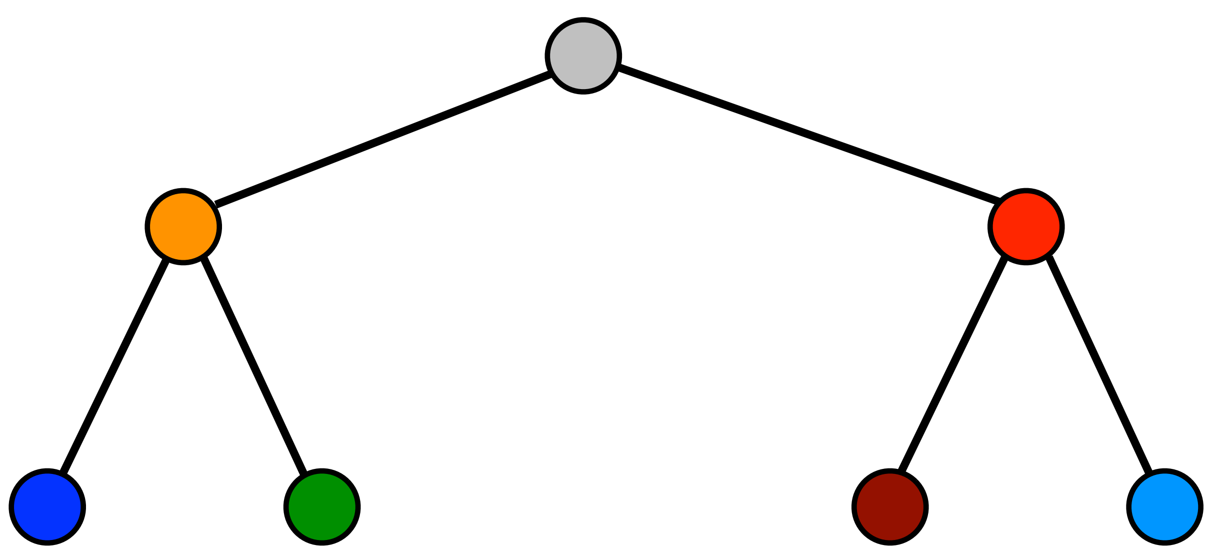




Example: parallel pointer chasing

```
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
  process(p->data);
}
```





Task scheduling points

- Certain constructs have task scheduling points in them (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 thread and begin executing another task in the task pool (*task switching)*
- At the completion of the task or 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
          process(item[i]);
}
```

- Eventually too many tasks are generated
- Tasks that is generating task is suspended and the task thatis 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*

taskprivate data

- No longer supported
- too expensive

Synchronization

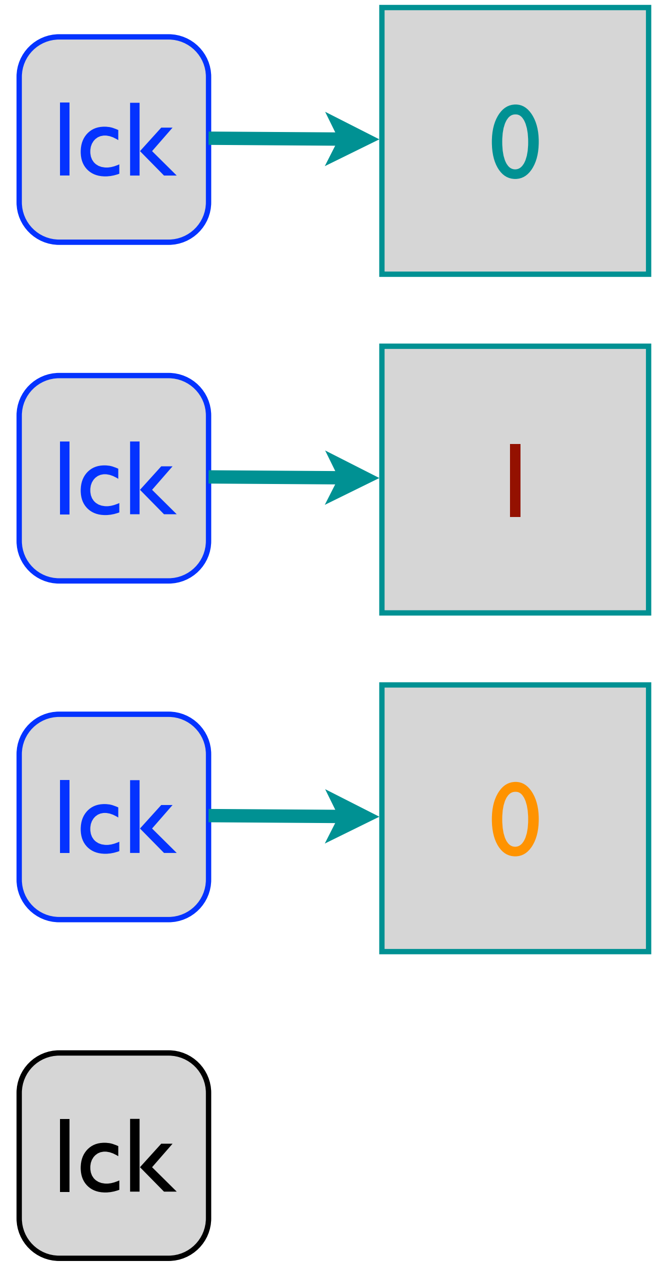
- Locks
- Nested locks

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

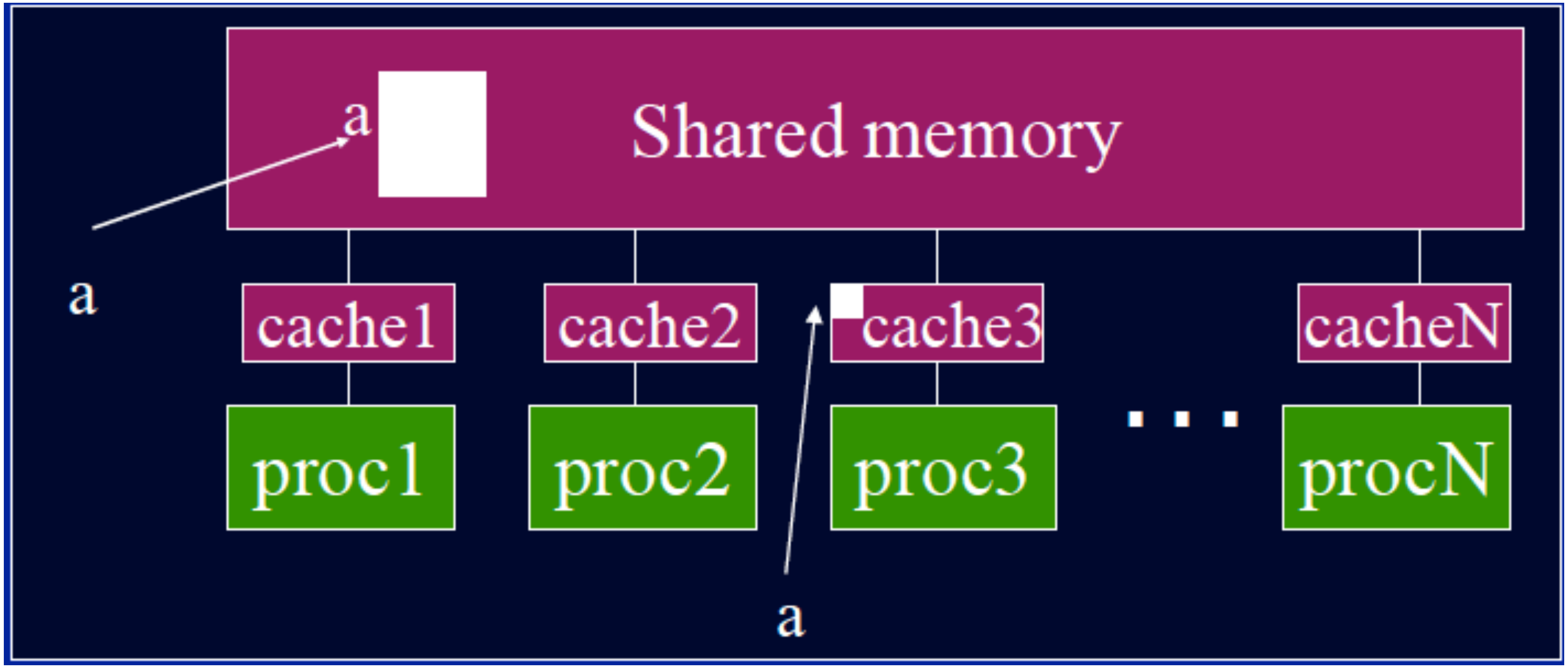
```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
  id = omp_get_thread_num();
  tmp = do_lots_of_work(id);
  omp_set_lock(&lck);
  printf("%d %d", id, tmp);
  omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```



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

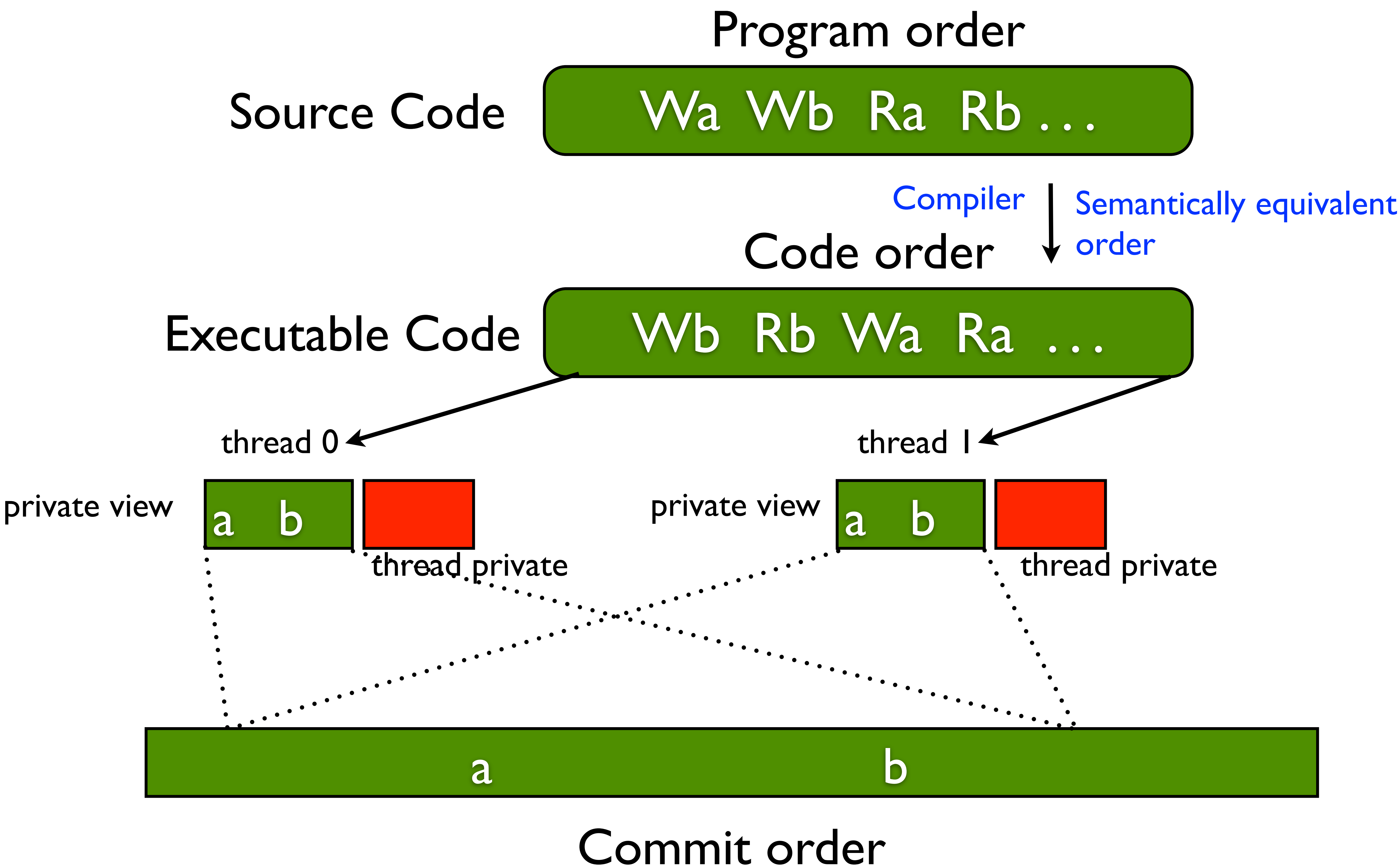


Two issues, coherence and consistency. *Coherence*: Behavior of the memory system when a single address is accessed by multiple threads. *Consistency*: Orderings of accesses to different addresses by multiple threads.

Memory models

- Memory models worry about the interactions of loads and stores (reads and writes) in different threads
- Dependences are used to deal with reads and writes within a thread and are not generally thought of as part of the memory model.
  - Stated differently, regardless to of the memory model, reads and writes within a thread will always honor dependences.

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.*

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*

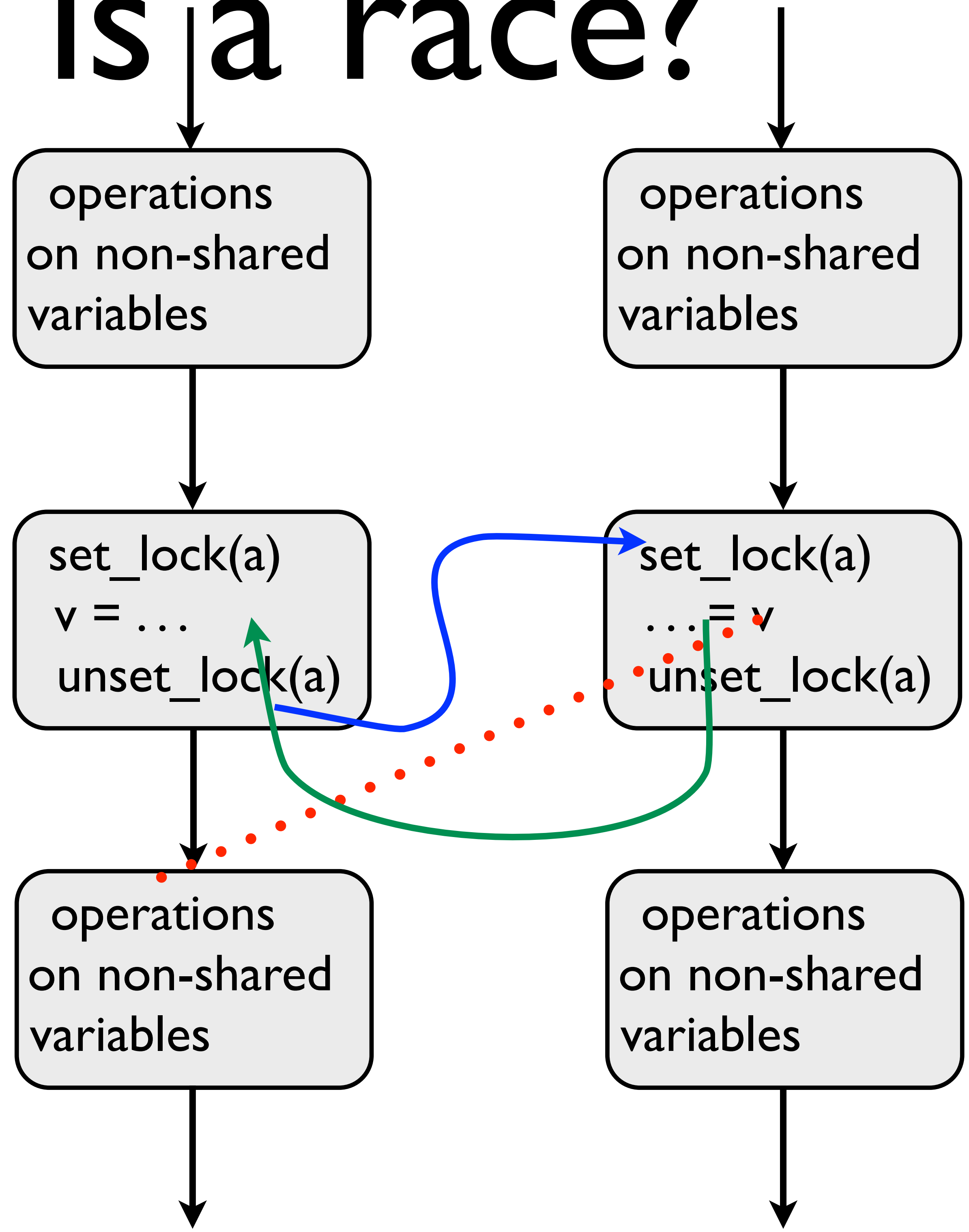
$$R \rightarrow R,\ W \rightarrow W,\ R \rightarrow W,\ W \rightarrow R,\ R \rightarrow S,\ S \rightarrow S,\ W \rightarrow S$$

OpenMP's consistency model

- Weak consistency
- S ops (synchronization) must be in sequential order
  - Within a thread cannot reorder S with W or S with R
  - Guarantees S→W, S→R, R→S, W→S, S→S
  - R→R, W→W, R→W missing. Obviously, if writes to the same location they are ordered (dependences enforced)

What is a race?

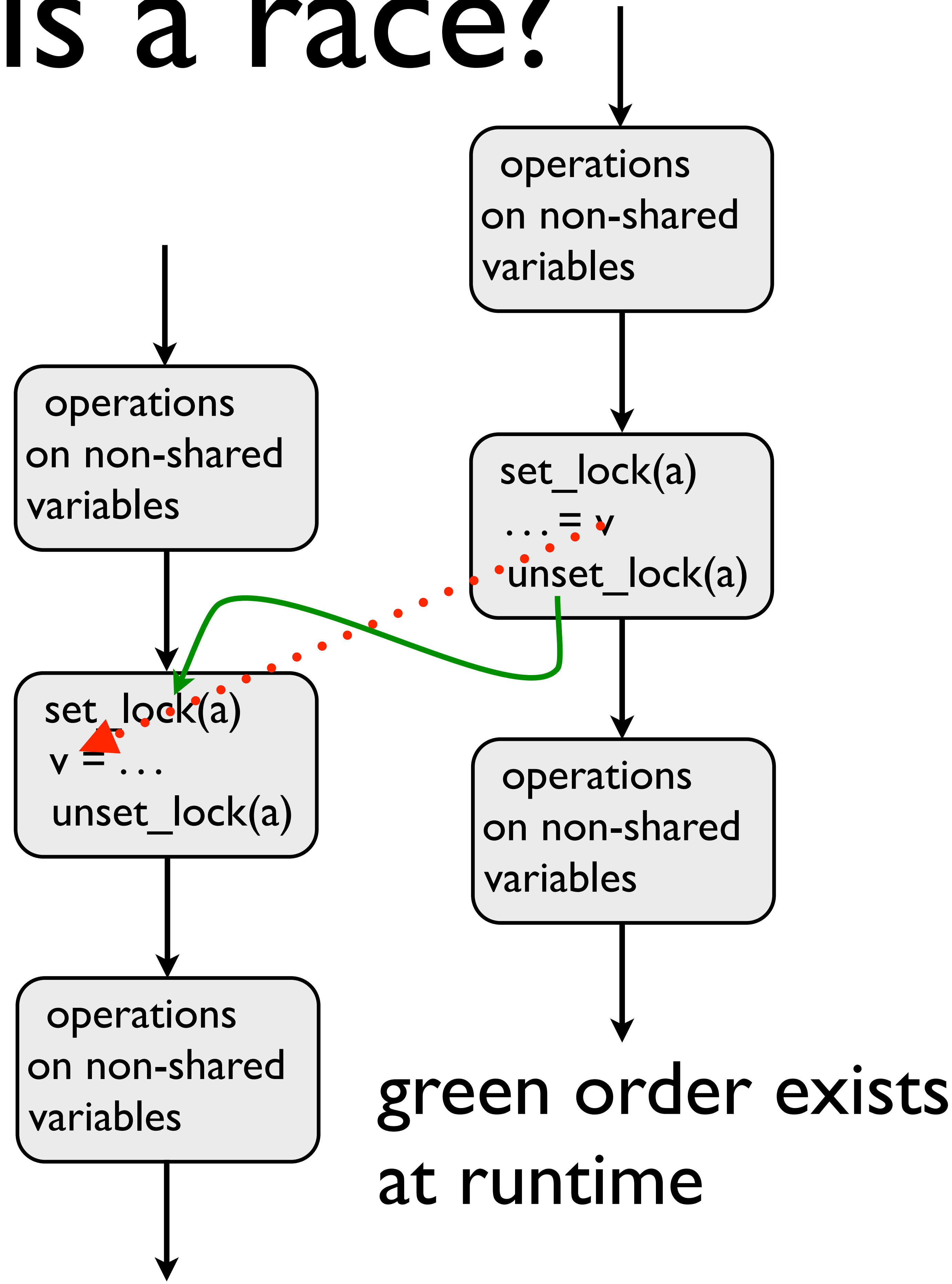
- Execute a parallel program
- If a there is a read or write to some v in a thread, and a write to it in another thread, and no enforced ordering at runtime between the two, there is a race.
- Orderings come from synchronization



either blue or green order *must* exist at runtime

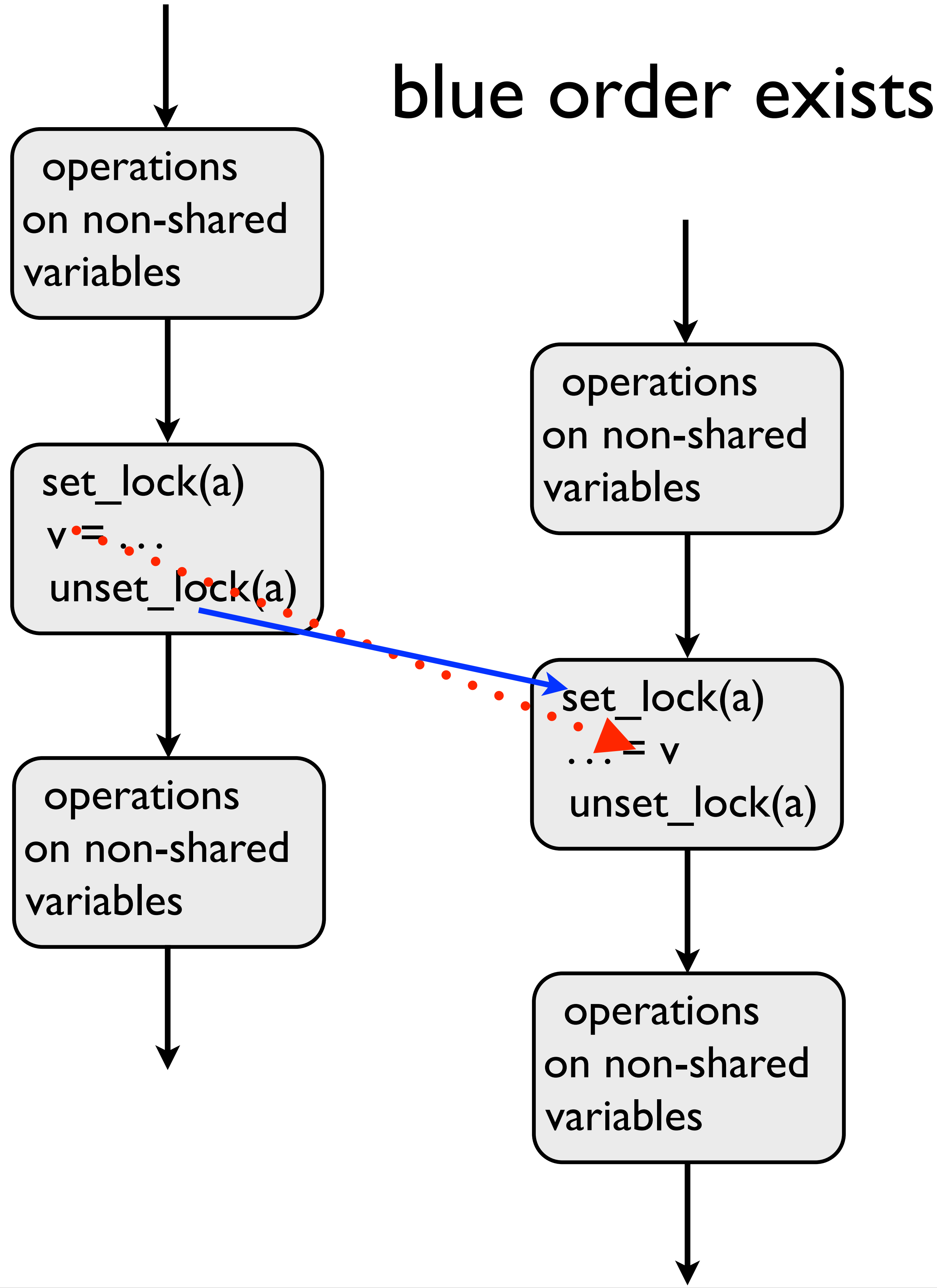
What is a race?

- 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



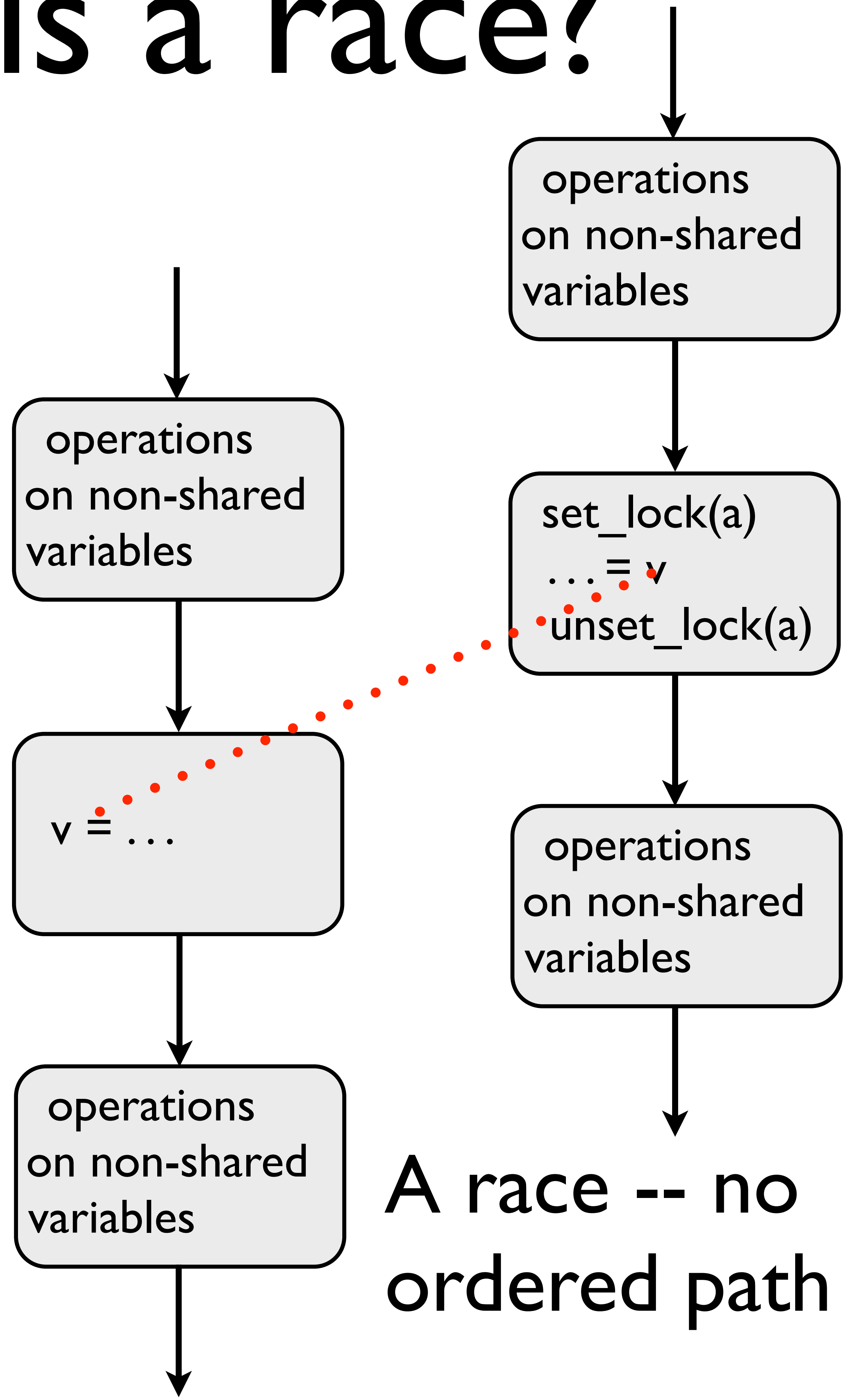
What is a race?

- 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



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



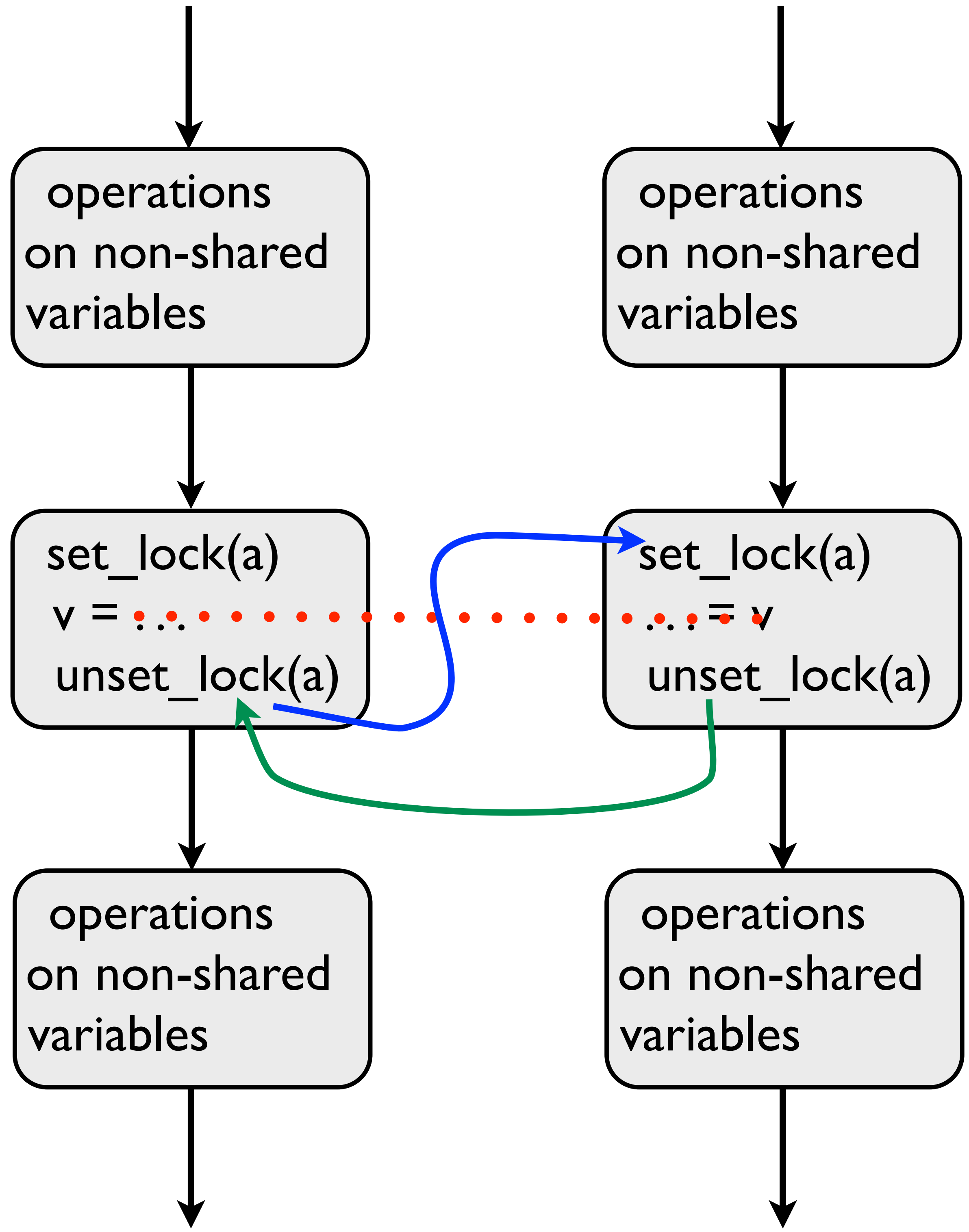
A race -- no ordered path exists

Back to yesterday's discussion

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

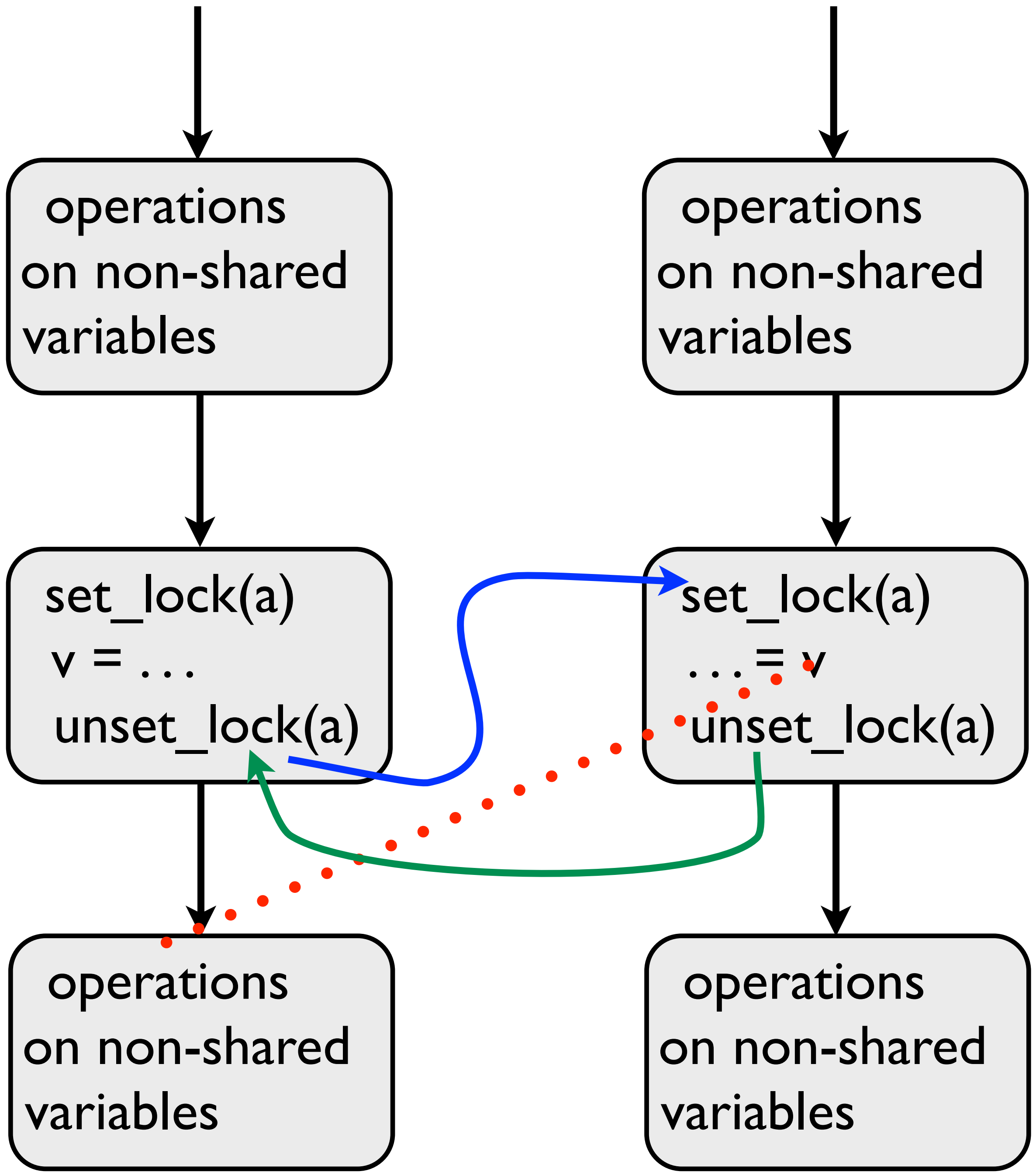
The value to memory will not be committed before any stale values of v are invalidated



What about 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 makes the machine faster (*sync*'s execute faster)



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
- *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
- 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
```

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 between threads -- the make local and shared memory consistent for a thread.

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

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

Environment variables and functions

- 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()
- *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( )