COMP 422, Lecture 8: Memory Consistency Models and Advanced OpenMP

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@rice.edu

Recap of Lecture 7 (OpenMP)

- Parallel regions
  - —master & worker threads
- Work-sharing constructs
  - —parallel loop w/ schedule clauses
  - —parallel sections
  - —Single
- Synchronization
  - —nowait
  - —barrier
- Data-sharing attributes
  - —private, firstprivate, lastprivate, shared, reduction, threadprivate
- Runtime routines
  - —omp_get_thread_num (), int omp_get_num_threads (), omp_in_parallel()
- Environment variables
  - —OMP_NUM_THREADS, OMP_SET_DYNAMIC, OMP_NESTED, OMP_SCHEDULE

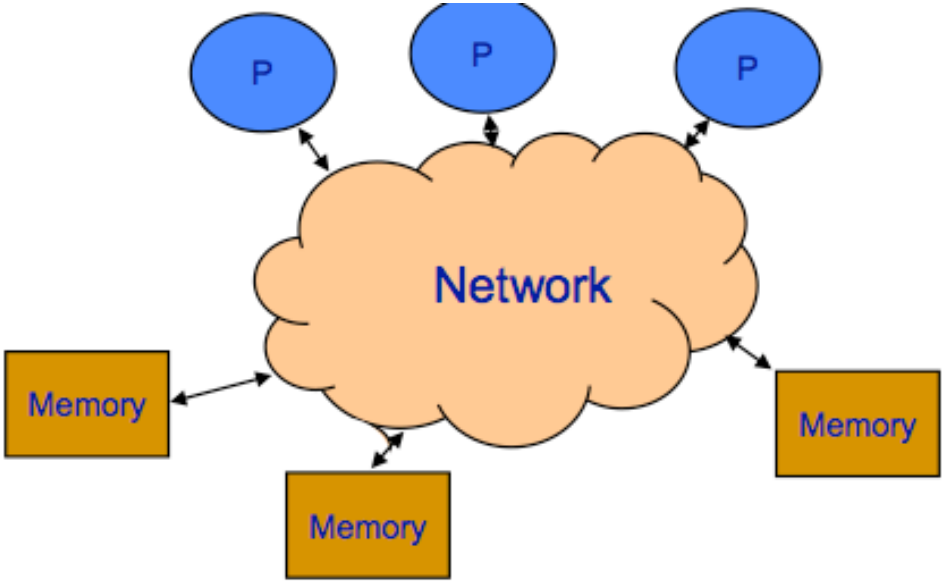
Acknowledgments for today's lecture

- "Memory Coherence and Consistency" lecture by Sudhakar Yalamanchili, Georgia Tech, ECE 4100/6100 course
  - http://www.ece.gatech.edu/academic/courses/fall2006/ece6100/Lectures/Module%2011%20-%20%20Coherence%20and%20Consistency/module.coherence.consistency.pdf
- "Memory Consistency Model" lecture by Sarita Adve, Fall 2001, http://www.cs.cornell.edu/Courses/cs717/2001fa/lectures/sarita.ppt
- "Memory Consistency Models: Convergence At Last!", presentation by Sarita Adve, December 2007
  - http://www.cse.iitk.ac.in/users/mtworkshop07/workshop/DayIII/adve.sarita.pdf
- "The OpenMP Memory Model" Jay Hoeflinger & Bronis de Supinski, IWOMP 2005 workshop, June 2005
  - http://www.compunity.org/events/pastevents/iwomp2005/hoeflinger.ppt
- OpenMP 2.5 specification
  - http://www.openmp.org/mp-documents/spec25.pdf
- "Towards OpenMP 3.0", Larry Meadows, HPCC 2007 presentation
  - http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc07_Larry.ppt
- OpenMP 3.0 draft specification
  - http://www.openmp.org/mp-documents/spec30_draft.pdf

Outline

- **Memory Consistency Models**
- **OpenMP Memory Model & flush operation**
  - **Sections 1.4 & 2.7.5 of OpenMP 2.5 spec**

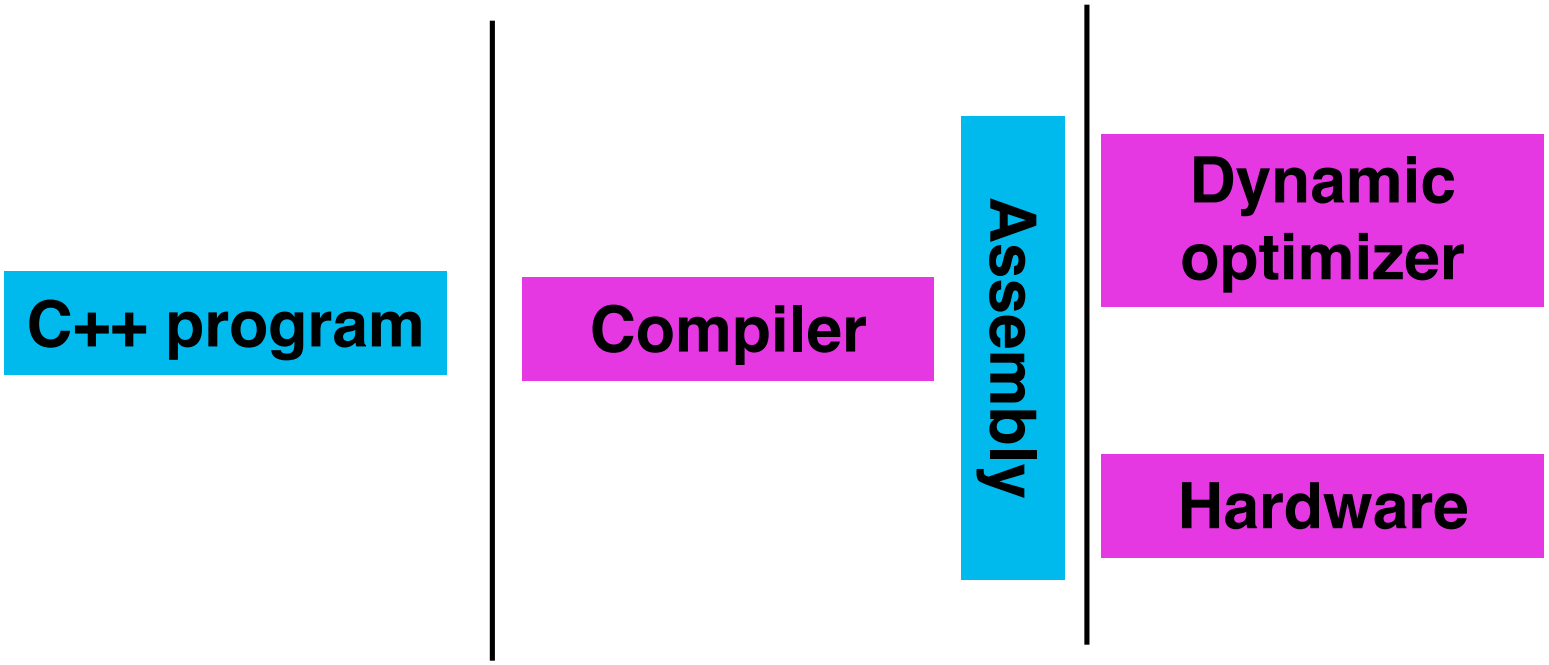
Memory Consistency Models



- A memory consistency model →
  - A set of rules governing how the memory systems will process memory operations from multiple processors
  - Contract between the programmer and system
  - Determines what optimizations can be performed for correct programs

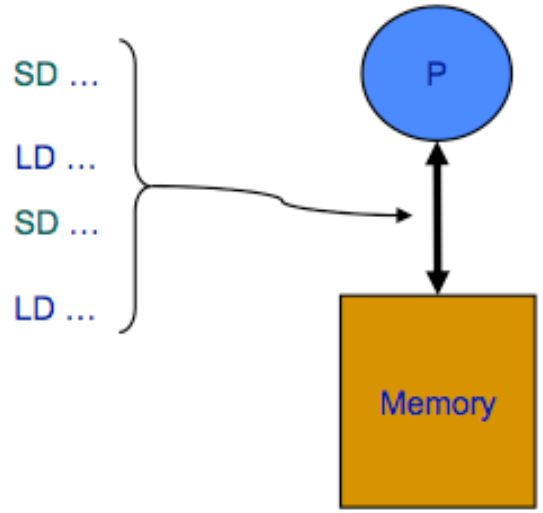
What is a Memory Model?

- Interface between program and transformers of program
  - —Defines what values a read can return



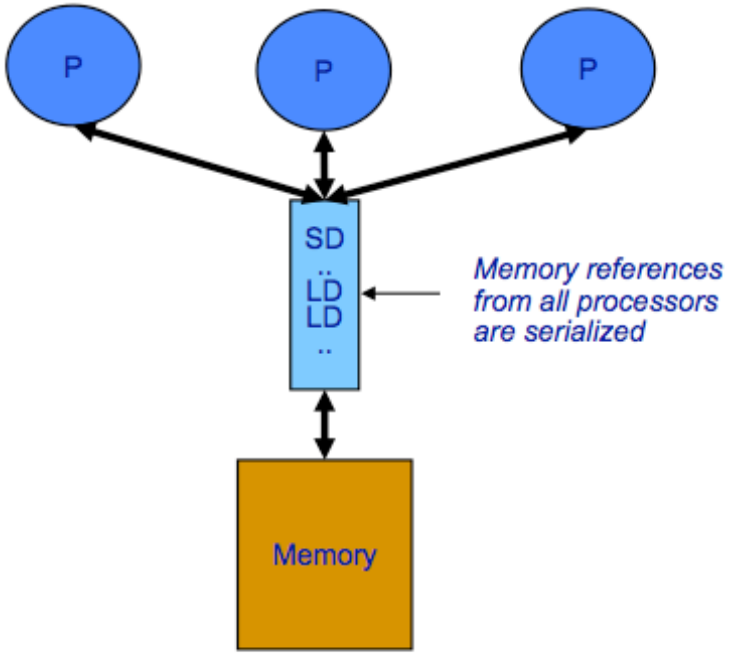
- Language level model has implications for hardware
- Weakest system component exposed to programmer

Uniprocessor Memory Model



- What is the model of memory behavior?
  - Memory operations occur in program order → read returns the value of the last write in program order
- Semantics defined by sequential program order
  - Simple to reason about but over constrained
    - Really only need to honor control and data dependencies
  - Reality is that *independent* operations *can* execute in parallel
  - Optimizations preserve these semantics

Sequential Consistency



[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

Sequential Consistency

- **SC constrains all memory operations:**
  - **Write → Read**
  - **Write → Write**
  - **Read → Read, Write**
- **Simple model for reasoning about parallel programs**
- **But, intuitively reasonable reordering of memory operations in a uniprocessor may violate sequential consistency model**
  - **Modern microprocessors reorder operations all the time to obtain performance e.g., write buffers, overlapped writes,non-blocking reads…**
  - **Optimizing compilers perform code transformations that have the effect of reordering memory operations e.g., scalar replacement, register allocation, instruction scheduling, …**
  - **A programmer may perform similar code transformations for software engineering reasons without realizing that they are changing the program's semantics**

Example of SC violation: Store Buffer

Introduction to Memory Consistency

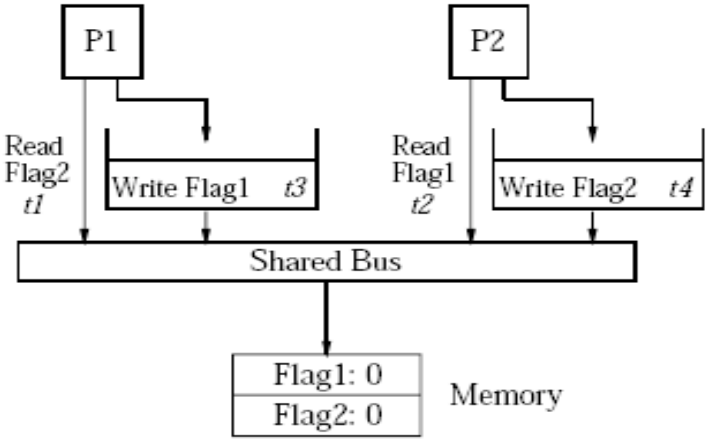
Example :

```
// Dekker's algorithm for critical sections
// Initially Flag1 = Flag2 = 0
```

P1	P2
`Flag1 = 1;  W(Flag1)`	`Flag2 = 1;  W(Flag2)`
`If (Flag2 == 0) R(Flag2)`	`if (Flag1 == 0) R(Flag1)`
`// critical section`	`// critical section`
`...`	`...`

Correct execution if a processor's Read operation returns 0 iff its Write operation occurred before both operations on the other processor.

- **Relaxed consistency : buffer write operations**
  - **Breaks Sequential Consistency**
  - **Invalidates Dekker's algorithm**
  - **Write operations delayed in buffer**



Material from & further reading: Adve, S. V. and Gharachorloo, K. 1996. Shared Memory Consistency Models: A Tutorial. Computer 29, 12 (Dec. 1996), 66-76. DOI= http://dx.doi.org/10.1109/2.546611

Weak Ordering

- Weak ordering:
  - Divide memory operations into data operations and synchronization operations
  - Synchronization operations act like a fence:
    - All data operations before synch in program order must complete before synch is executed
    - All data operations after synch in program order must wait for synch to complete
    - Synchs are performed in program order
  - Hardware implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed

Release Consistency (RC)

- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Release: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
  - but accesses after release in program order do not have to wait for release
  - operations which follow release and which need to wait must be protected by an acquire

Definition of a Data Race

- Only need to define for SC executions ⇒ total order
- Two memory accesses form a race if
  - —From different threads, to same location, at least one is a write
  - —May execute consecutively in an SC global total order i.e., may execute "in parallel"

Thread 1	Thread 2
Write, A, 26	
Write, B, 90	
	Read, Flag, 0
Write, Flag, 1	
	Read, Flag, 1
	Read, B, 90

- Data-race-free-program = No data race in any SC execution

  ==> Your goal should be to write only data-race-free parallel programs!

Defining Memory Consistency at the Programming Level: Data-Race-Free-0

- Different operations have different semantics

P1	P2
A = 23;	while (Flag != 1) {;}
B = 37;	… = B;
Flag = 1;	… = A;

- Flag = Synchronization; A, B = Data
- Can reorder data operations
- Distinguish data and synchronization
- Need to
  - Characterize data / synchronization
  - Prove characterization allows optimizations while preserving SC semantics

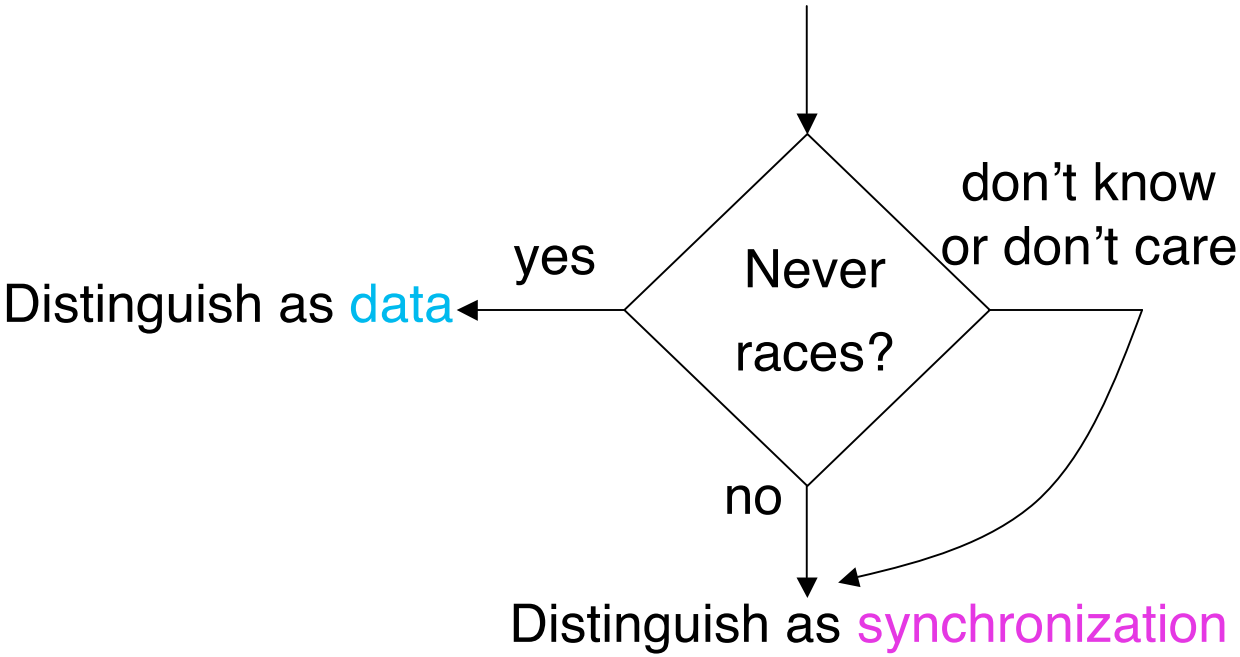
Data-Race-Free-0 (DRF0) Definition

- **Data-Race-Free-0 Program**
  - —All accesses distinguished as either synchronization or data
  - —All races distinguished as synchronization
    (in any SC execution)
- **Data-Race-Free-0 Model**
  - —Guarantees SC to data-race-free-0 programs
  - —(For others, reads return value of some write to the location)

Programming with Data-Race-Free-0

- **Information required:**
  - ***This operation never races* (in any SC execution)**

1. **Write program assuming SC**
2. **For every memory operation specified in the program do:**



Distinguishing Data vs. Sync Memory Operations in a Programming Model

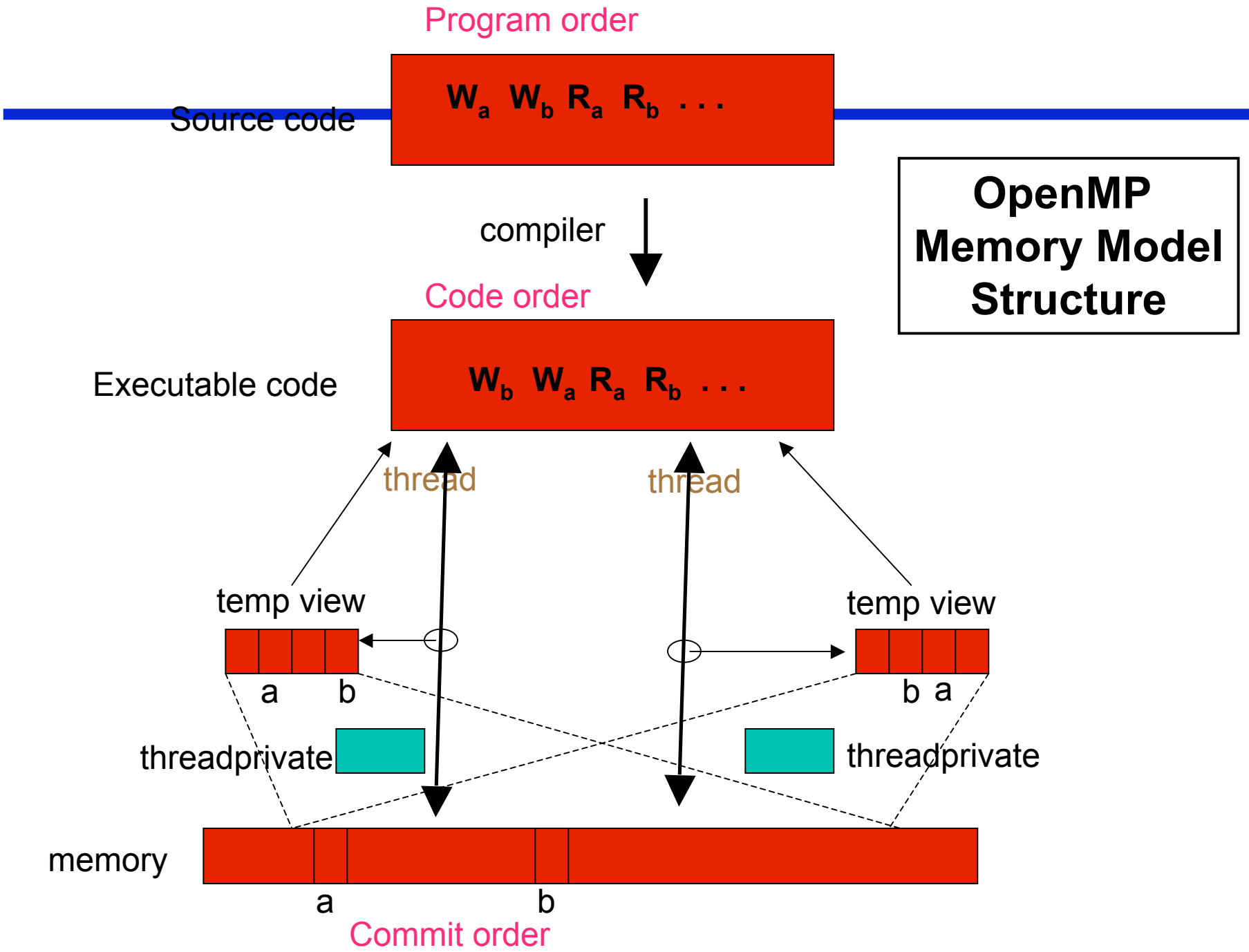
- **Option 1: Annotations at statement level**
  - P1
  - data = ON
    A = 23;
    B = 37;
  - synchronization = ON
    Flag = 1;

  P2
  synchronization = ON
  while (Flag != 1) {;}
  data = ON
  … = B;
  … = A;

- **Option 2: Declarations at variable level**
  - synch int: Flag
  - data int: A, B
- **Option 3: Treat flush/barrier operations as synchronization statements**
  - synchronization = ON
  - flush
  - synchronization = OFF

Outline

- Memory Consistency Models
- OpenMP Memory Model & flush operation
  - Sections 1.4 & 2.7.5 of OpenMP 2.5 spec
- Cilk_fence() operation
- OpenMP 3.0 tasks



OpenMP Memory Model Structure

Shared and Private Access

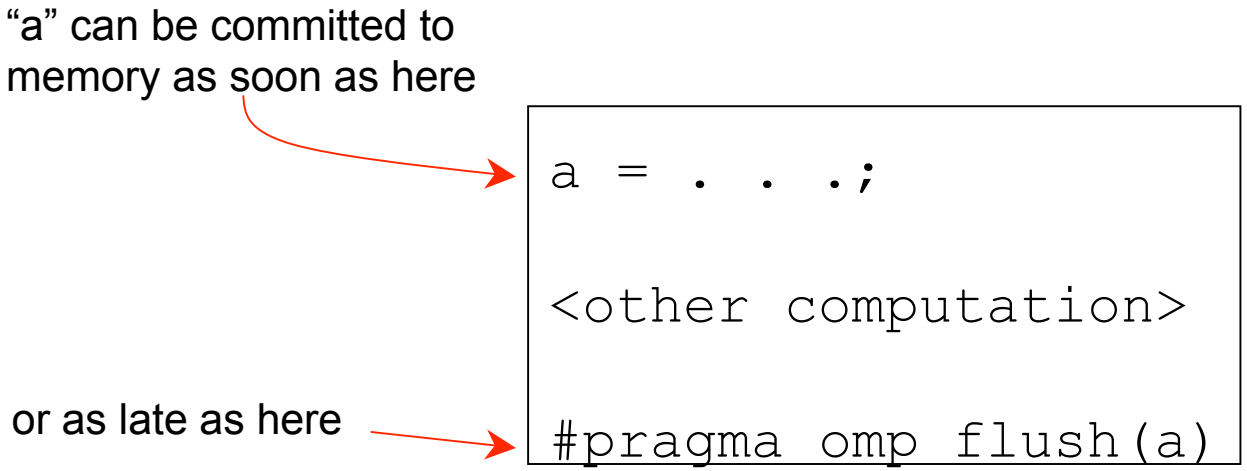
- All shared and private variables have original variables
- Shared access to a variable:
  - —Within the structured block, references to the variable all refer to the original variable
- Private access to a variable:
  - —A variable of the same type and size as the original variable is provided for each thread

Flush Is the Key OpenMP Operation

#pragma omp flush [(list)]

- **Prevents re-ordering of memory accesses across flush**
- **Allows for overlapping computation with communication**
- **A flush** construct with a list applies the flush operation to the items in the list,and does not return until the operation is complete for all specified list items.
  - —If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers
- A **flush** construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed.

Temporary View Allows Hiding Memory Latency



Re-ordering Example

```
a = ...;   //(1)
b = ...;   //(2)
c = ...;   //(3)

#pragma omp flush(c)    //(4)
#pragma omp flush(a,b)  //(5)

. . . a . . . b . . .; //(6)
. . . c . . .;         //(7)
```

(1) and (2) may not be moved after (5).

(6) may not be moved before (5).

(4) and (5) may be interchanged at will.

Moving data between threads

- To move the value of a shared var from thread a to thread b, do the following in exactly this order:
  - —Write var on thread a
  - —Flush var on thread a
  - —Flush var on thread b
  - —Read var on thread b

Example A.18.1 (OpenMP 2.5 spec)

```
/*  Announce that I am done with my work. The first flush
 *  ensures that my work is made visible before synch.
 *  The second flush ensures that synch is made visible.
 */
 #pragma omp flush(work,synch)
 synch[iam] = 1;
 #pragma omp flush(synch)

/* Wait for neighbor. The first flush ensures that synch is read
  * from memory, rather than from the temporary view of memory.
  * The second flush ensures that work is read from memory, and
  * is done so after the while loop exits.
  */
 neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
 while (synch[neighbor] == 0) {
   #pragma omp flush(synch)
 }
 #pragma omp flush(work,synch)
 /* Read neighbor's values of work array */
 result[iam] = fn2(work[neighbor], work[iam]);
```

Implicit flushes

- In barriers
- At entry to and exit from
  - Parallel, parallel worksharing, critical, ordered regions
- At exit from worksharing regions (unless `nowait` is specified)
- In `omp_set_lock`, `omp_set_nest_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock`
- In `omp_test_lock`, `omp_test_nest_lock`, if lock is acquired
- At entry to and exit from `atomic` - flush-set is the address of the variable atomically updated

OpenMP ordering vs. weak ordering

- **OpenMP re-ordering restrictions are analogous to weak ordering with "flush" identified as a "synch" op.**
- **But, it's weaker than weak ordering.**
  - **—Synchronization operations on disjoint variables are not ordered with respect to each other in OpenMP**

Relaxed memory model enables use of NUMA machines e.g., clusters and accelerators

Importance of variable list in flush

Incorrect example:

```
a = b = 0
```

thread 1	*thread 2*
`b = 1`	`a = 1`
flush`(b)`	*flush*`(a)`
flush`(a)`	*flush*`(b)`
`if (a == 0) then`	`if (b == 0) then`
critical section	*critical section*
`end if`	`end if`

Correct example:

```
a = b = 0
```

thread 1	*thread 2*
`b = 1`	`a = 1`
flush`(a,b)`	*flush*`(a,b)`
`if (a == 0) then`	`if (b == 0) then`
critical section	*critical section*
`end if`	`end if`