Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 9:

- ❑ Shared-Memory Programming with OpenMP
  - Memory Models
  - OpenMP Memory Model

Memory Models

Memory models define the allowed behavior of a shared memory system in terms of the order in which **memory reads** and **memory writes** may appear to execute.

- The memory model specifies, for a given multithreaded program, what values a read can return.
  - There can be more than one possible outcomes allowed by the given memory model.
- The memory model serves as an interface between the programmer and the system so it
  - affects **programmability** because programmers must use it to reason about the correct of their programs
  - affects **performance** because it determines the types of optimizations that may be exploited by the hardware and the system software
  - affects **portability** when moving software across different systems that support different memory models

Memory Models

A **memory model** needs to be specified **for every level** at which there is an interface between the programmer and the system:

- At the **ISA** interface, the memory model affects the designer of the computer architecture and the programmer who writes or reasons about machine code that runs on the hardware
- At the **high-level language** interface, the memory model affects both the designer of the compiler software and the programmer who writes or reasons about code written in high-level languages.

Therefore, the **memory model** has a significant impact on

- the writing of multithreaded programs from the programmer’s perspective and
- virtually all aspects of designing a parallel system(including processor, memory system, interconnection network, compiler and programming languages).

Memory Models

Reasoning about multithreaded execution can be mind-boggling.

Assume that x = 0 and y = 0, initially.

```
Thread T0                Thread T1
  x = 1                    r1 = y
  y = 1                    r2 = x
```

One would expect that only the following three **final states** (r1,r2) are possible: (0,0), (0,1) and (1,1)

- Q: What is wrong with (r1,r2)=(1,0)?
- A: We would probably think this final state wouldn’t be reached because our intuition is based on the **Sequential Consistency** model.

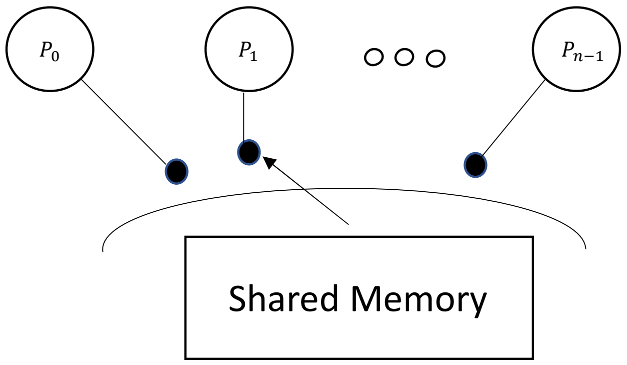
Sequential Consistency Model

The most intuitive memory consistency model is the **Sequential Consistency** (SC) model.

- SC was first formalized by Lamport in **[1]**:

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

- The behavior of SC should be the same as if there were **a single global switch** that multiplexes among memory operations from the processors.



Sequential Consistency Model

There are two key aspects to **Sequential Consistency**:

- Maintaining **program order** among operations from **individual threads**
- Maintaining a **single sequential order** among operations from **all threads**
  - referred to as **write atomicity** makes it appear as if each memory operation executes **atomically** or **instantaneously** with respect to other memory operations.

Sequential Consistency Model

Sufficient Conditions for SC

- ❑ **Program Order**: memory ordering has to follow the order in each individual thread
  - Threads issue memory operations in **program order**.
  - Before issuing the next memory operations, threads wait until their last issued memory operations complete.
- ❑ **Write Atomicity**: a store by a thread is **logically seen** by all other threads at once
  - A read is allowed to complete only if the matching write (i.e., the one whose value is returned by the read) also **completes**(i.e., **until the write becomes visible to all threads**).

In a SC system, the operations of a program **appear to take place** in **some total order**, and that total order is consistent with the order of operations on each individual thread.

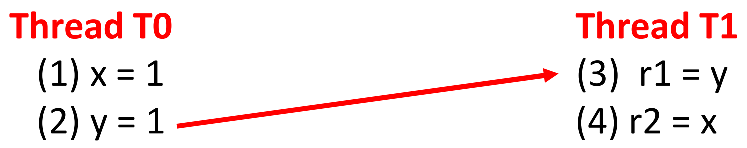
- SC is too strong.
- The two conditions are easily violated by **modern hardware** and **compilers**.

Sequential Consistency Model

Let us return to the previous example, where **x = y = 0**, initially.

Suppose that **r1=1** and **r2 = 0**.

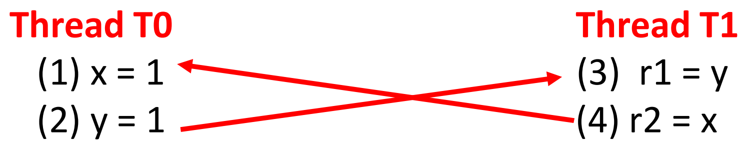
- **r1=1** implies (2) **happens-before** (3):



Thread T0
(1) x = 1
(2) y = 1

Thread T1
(3) r1 = y
(4) r2 = x

- **r2=0** implies (4) **happens-before** (1):



Thread T0
(1) x = 1
(2) y = 1

Thread T1
(3) r1 = y
(4) r2 = x

Sequential Consistency Model

By the **Program Order** condition,

- (1) **happens-before** (2)
- (3) **happens-before** (4)

$$\begin{tikzcd}
\textbf{Thread T0} & & \textbf{Thread T1} \\
(1)\ x = 1 \arrow[d] & & (3)\ r1=y \arrow[d] \\
(2)\ y = 1 \arrow[urr] & & (4)\ r2 = x \arrow[ull]
\end{tikzcd}$$

We can see that the happens-before relationship is cyclic.

- This final state cannot be reached in SC.

Sequential Consistency Model

- ❑ The rules of SC we just discussed are **intuitive to the programmer**.
  - The idea is that multiple threads executing in parallel are accessing **a single shared main memory**.

- ❑ SC implementations will result in **poor performance**.
  - SC is a strong memory model.
  - The benefits of multiple threads, instruction-level parallelism, etc. are greatly limited ☹

Weak Memory Model

Dekker's Algorithm

Assume that **Flag1 = Flag2 = 0**, initially.

```
Thread T0                         Thread T1
(1) Flag1 = 1                     (3) Flag2 = 1
(2) if(Flag2==0)                  (4) if(Flag1==0)
        Critical Section                  Critical Section
```

Q: Can the two threads be inside the critical section simultaneously?

A: It depends.

- ❑If Dekker's Algorithm runs on an SC machine, **no**.
- ❑Otherwise, **yes**, on a machine with a weaker memory model.

Weak Memory Model: Write Buffer

On most real hardware, what causes **Dekker's Algorithm** to break is the existence of **write buffers**.

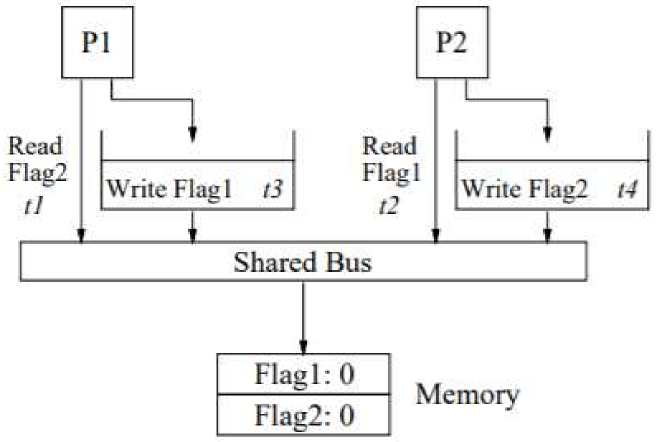


Figure Credit: [2]

Weak Memory Model: Write Buffer

Purposes of the write buffer

- ❑ If the core issues writes that occur at a faster rate than the cache can respond, the write buffer can absorb the pending writes while the core can proceed to execute other instructions without stalling.

- ❑ If there are multiple writes to **different words** that reside on **the same cachline**, these writes can be combined into a single write, which can reduce the number of writes that need to be serviced by the cache.

Weak Memory Model: Write Buffer

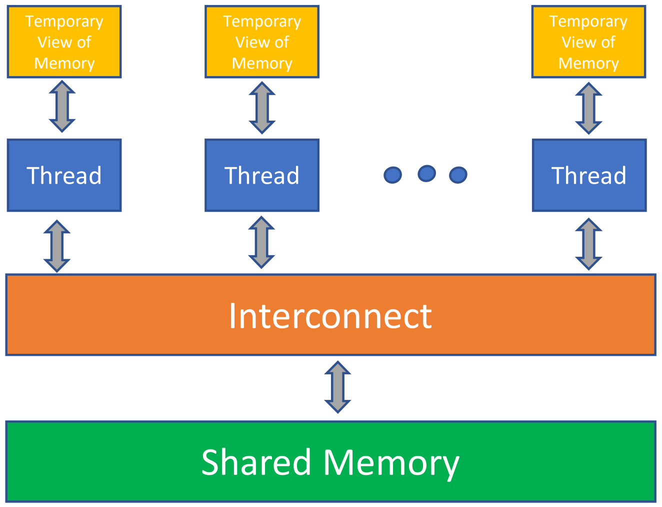
- ❑ On a write, a processor simply inserts the write operation into the write buffer and proceeds without waiting for the write to complete.
- ❑ Subsequent reads are allowed to **bypass** any previous writes in the write buffer for faster completion.
  - This bypassing mechanism is allowed as long as the read address does not match any of the pending writes in the write buffer.
  - In the case of **Dekker's Algorithm**, both reads can be serviced by the memory system before either write is serviced, thereby allowing both reads to return 0.

OpenMP: Memory Model

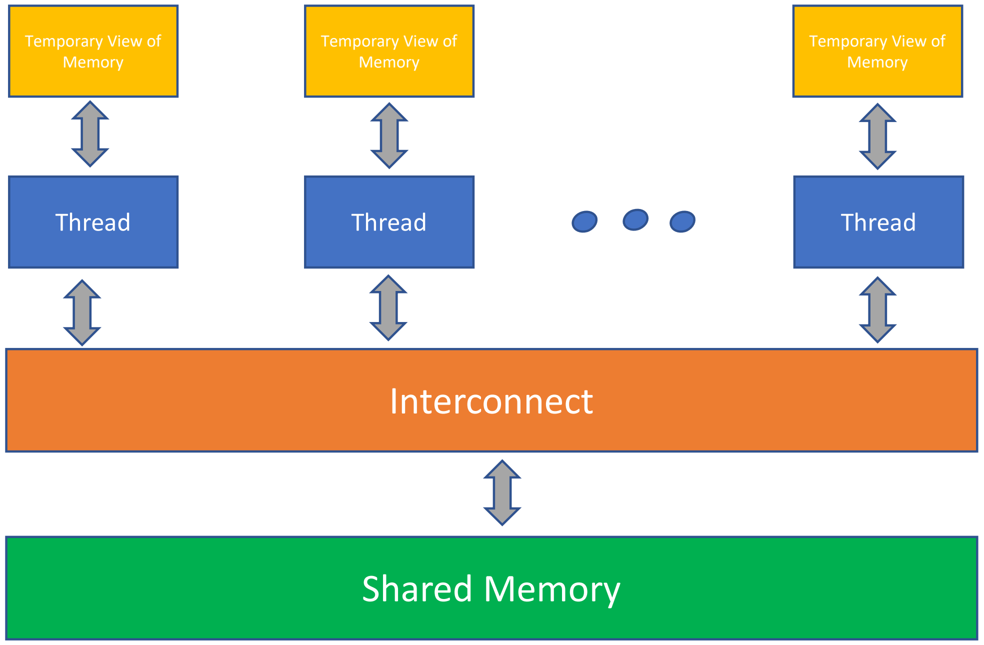
Temporary view and Memory

The OpenMP memory model consists of two components:

- ❑ The **global memory** shared by all threads (processors)
- ❑ The **temporary view** of the memory associated with **the individual threads**



OpenMP: Memory Model



- ❑ Read and write operations of a thread refer to **the temporary view**.
  - The temporary view may or may not be consistent with the shared memory.
- ❑ Consistency can only be achieved by a **flush** operation.
  - A **flush** ensures that whatever a thread has **written** to its own temporary view will be **visible** in the shared memory.
  - Whatever a different thread has flushed to the shared memory will be **available** for **reading** in the thread's temporary view.

OpenMP: Memory Model

OpenMP supports a **relaxed memory model**.

- ❑ Each thread can maintain a **temporary view** of shared memory that is not consistent with that of other threads.
- ❑ These temporary views are made consistent with the shared memory **at certain points** in the program.
- ❑ The operation used to maintain consistency is the **flush** operation.

OpenMP: Flush

Each thread has a **temporary view** of **shared memory** that it uses to store data temporarily and hidden from the other threads.

- ❑ Writes to memory can be delayed and overlapped with other operations.
- ❑ Reads from memory may be directly satisfied with local copies (cached in **CPU registers**) until they are forced into memory by **flushes**.

The **flush** operation defines a sequence of points at which a thread is guaranteed to see a **consistent view** of the shared memory.

- ❑ Upon a **flush**, a thread performs the following:
    - Force all the pending writes into the shared memory
    - Discard the local copies and satisfy reads directly from the shared memory
    - Prevent any reads or writes from this thread from reordering **across the flush**
        - This is analogous to a **fence** in other shared memory APIs/ISAs.

OpenMP: Flush

- ❑ Compilers can reorder instructions that implement a program.
  - Modern compilers are **highly aggressive** in that they generate code to exploit the functional units, keep the CPU busy, hide memory latencies etc.
  - The compiler generates code that maintains only the **as-if-serial semantics**.
    - When it comes to code for multithreaded programs, things can break.
- ❑ Compilers generally cannot move instructions past:
  - a barrier
  - a flush **on all variables**
- ❑ The flush operation does not synchronize different threads
  - It just ensures that a thread's variables (**temporary view**) are main consistent with the shared memory.

OpenMP: Flush

In OpenMP, a **flush** is **implicit** in the following synchronization locations:

- At entry to and exit from parallel regions
- At implicit and explicit barriers
- At entry to and exit from critical regions
- At exit from work-sharing constructs unless **nowait** is present
- Immediately before and after TSPs
- During the following runtime functions: **omp_set_lock()** and **omp_unset_lock()**
- At entry to and exit from the atomic construct

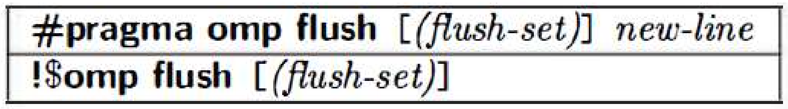
Warning

Avoid using explicit flushes !!!

- it is difficult to get it working .
- Even an expert often get it wrong.
- You may need explicit flushes if you want to implement your own spinlocks.

OpenMP: Flush

- The flush directive can take a **flush-set** specified with a list of variables a flush is to performed on.



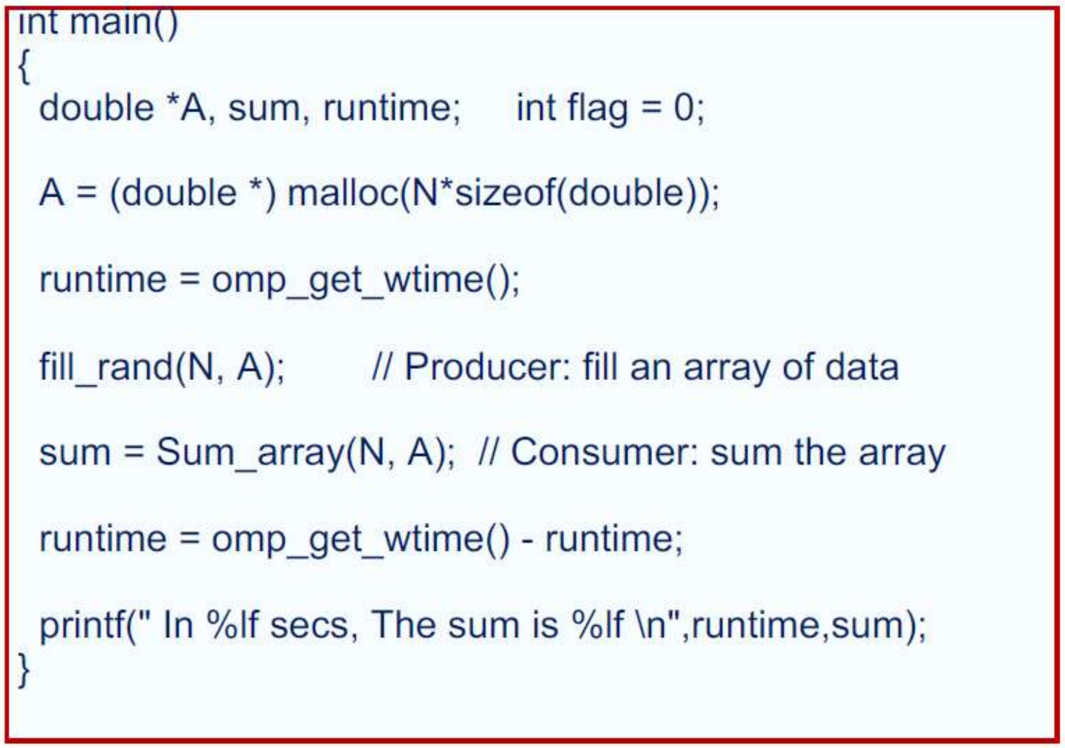
#pragma omp flush [(*flush-set*)] *new-line*
!$omp flush [(*flush-set*)]

- A flush without a **flush-set** is equivalent to one with a **flush-set** specified **with all visible variables**.

OpenMP: Flush

- ❑ The compiler is not allowed to reorder reads or writes of the variables in the **flush-set** around a flush.
  - All previous reads and writes of the **flush-set** by this thread have completed.
  - No subsequent reads or writes of the **flush-set** by this thread have occurred.
  - All variables in the **flush set** are moved from temporary storage, i.e., registers, to the shared memory.
  - Reads of variables in the **flush-set** that follow that flush are loaded from the shared memory.
- ❑ Again, keep in mind that the flush operation only makes the calling thread's temporary view consistent with the shared memory.
  - The flush operation by itself does not synchronize itself with other threads.

OpenMP: Flush



```
int main()
{
  double *A, sum, runtime;     int flag = 0;

  A = (double *) malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);        // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %lf secs, The sum is %lf \n",runtime,sum);
}
```

Producer-Consumer Problem

TO DO:

- Implement pairwise synchronization between the producer and the consumer thread

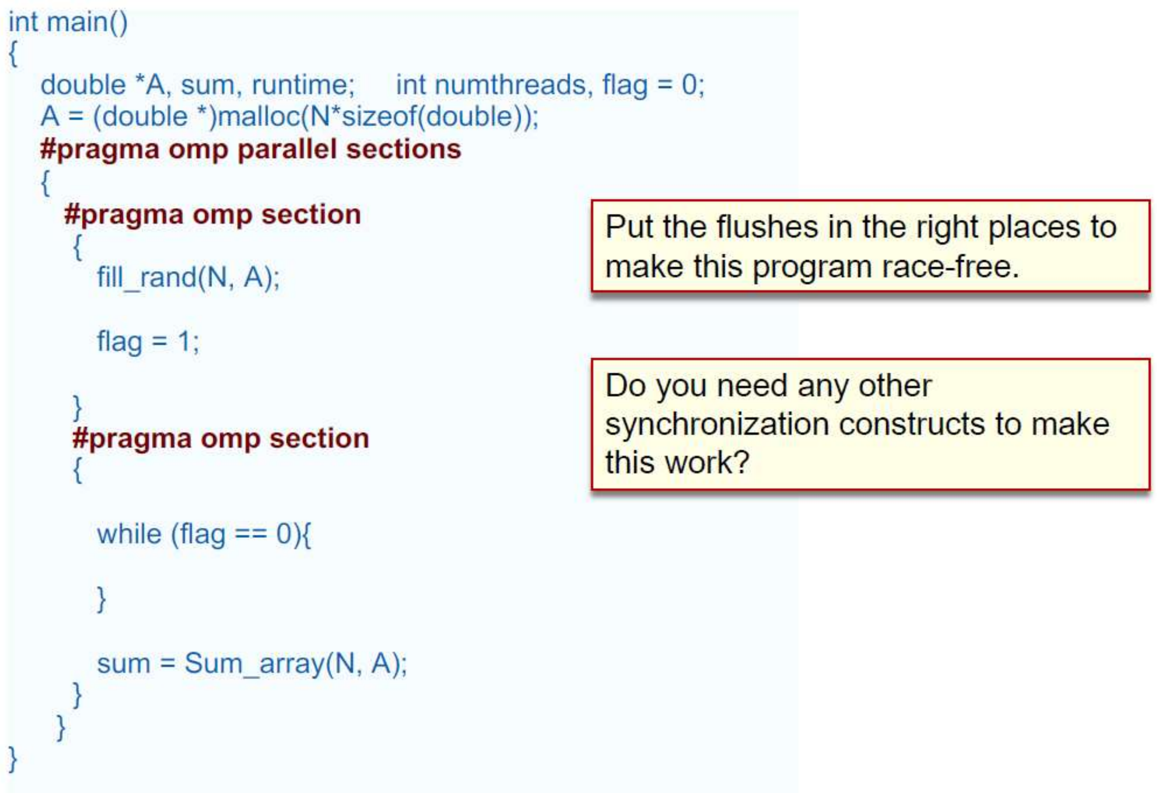
Figure Credit: [3]

OpenMP: Flush

- ❑ OpenMP lacks **pairwise constructs** that work between pairs of threads.
  - In Pthreads, we have **condition variables** that can be used to synchronize between a pair of threads.
- ❑ When needed, we have to implement it ourselves !!!

- ❑ Pairwise Synchronization can be implemented as follows:
  - Use a shared flag variable
  - Have the consumer spin waiting for the new value of the shared flag
  - Use flushes to force updates to and from memory

OpenMP: Flush



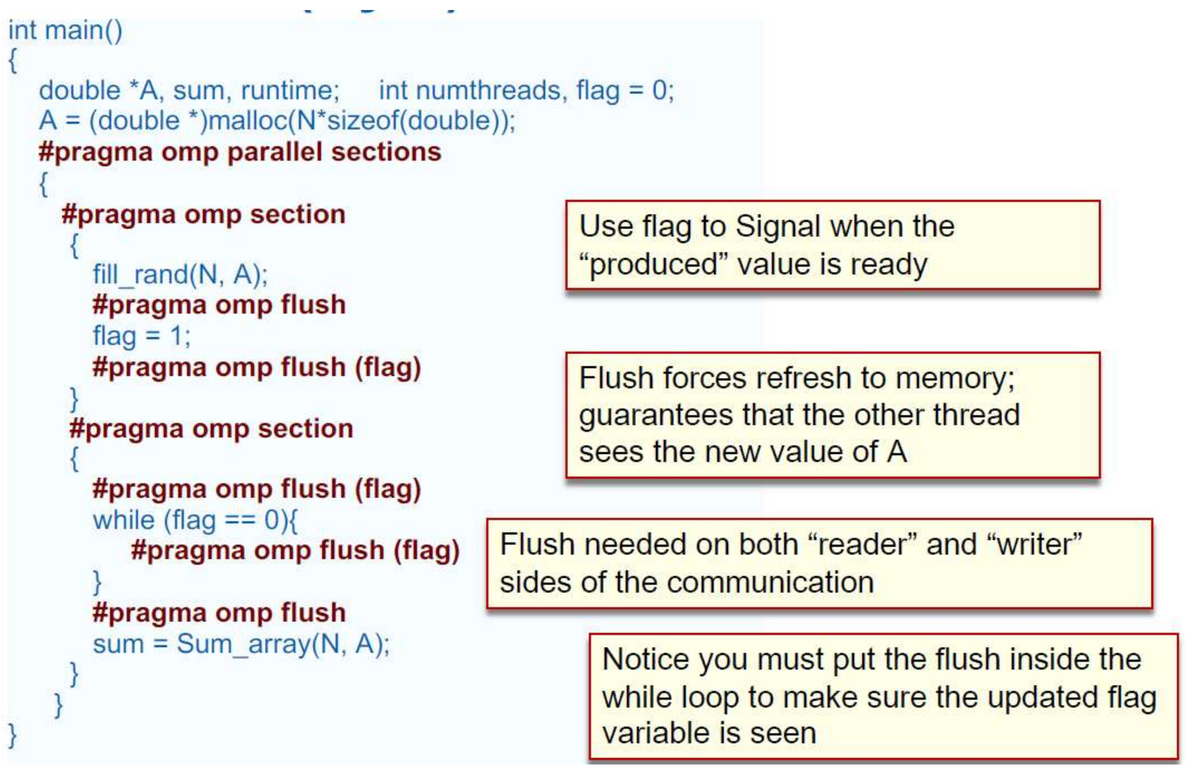
```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);

        flag = 1;

      }
      #pragma omp section
      {

        while (flag == 0){

        }

        sum = Sum_array(N, A);
      }
    }
}
```

Put the flushes in the right places to make this program race-free.

Do you need any other synchronization constructs to make this work?

Figure Credit: [3]

OpenMP: Flush



```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
      }
     #pragma omp section
      {
        #pragma omp flush (flag)
        while (flag == 0){
            #pragma omp flush (flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);
      }
   }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory; guarantees that the other thread sees the new value of A

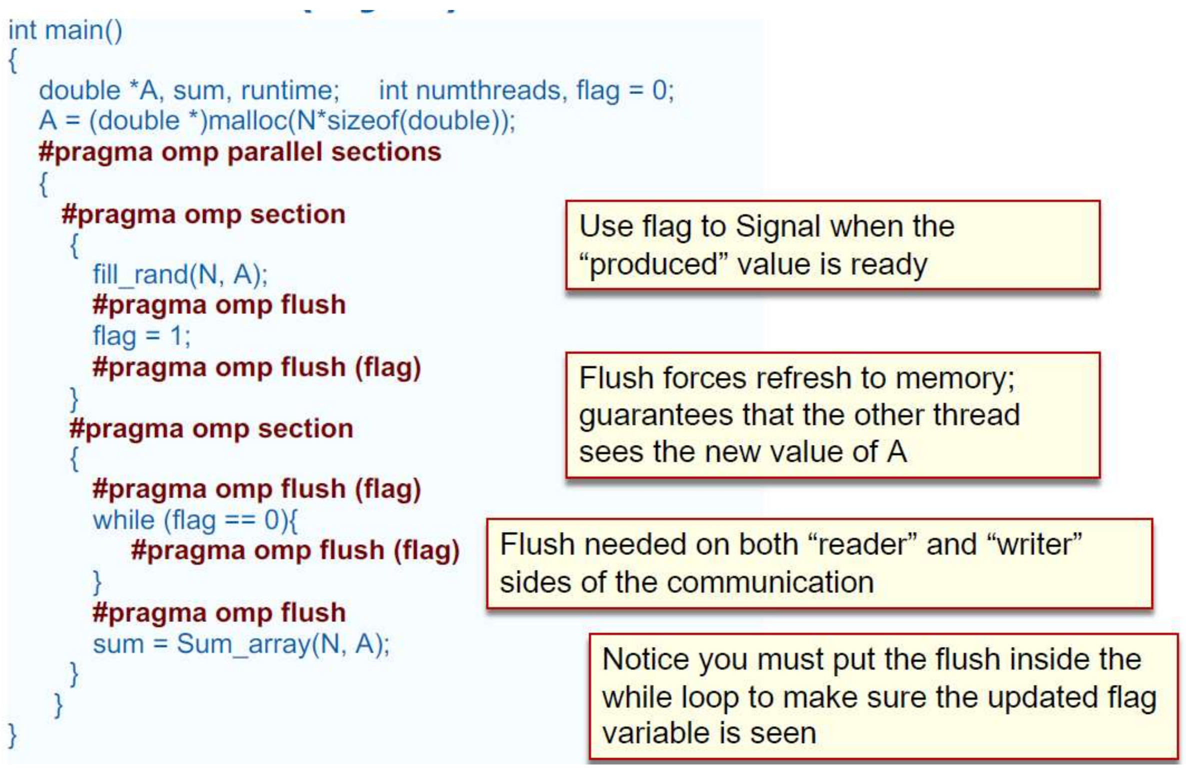
Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

Is this code correct?

Figure Credit: [3]

OpenMP: Flush



```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
      }
     #pragma omp section
      {
        #pragma omp flush (flag)
        while (flag == 0){
            #pragma omp flush (flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);
      }
    }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory; guarantees that the other thread sees the new value of A

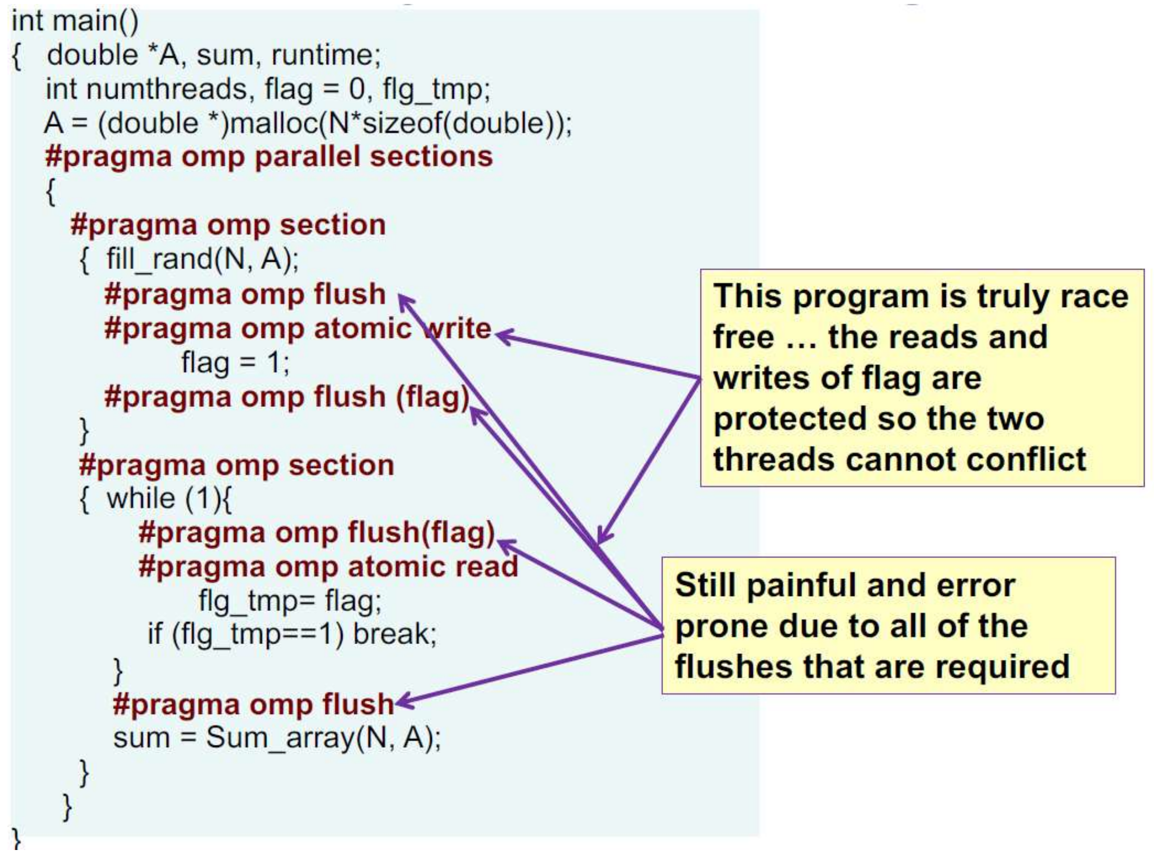
Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

This code is not **data-race free**, but it works correctly on x86 machines.

Figure Credit: [3]

OpenMP: Flush



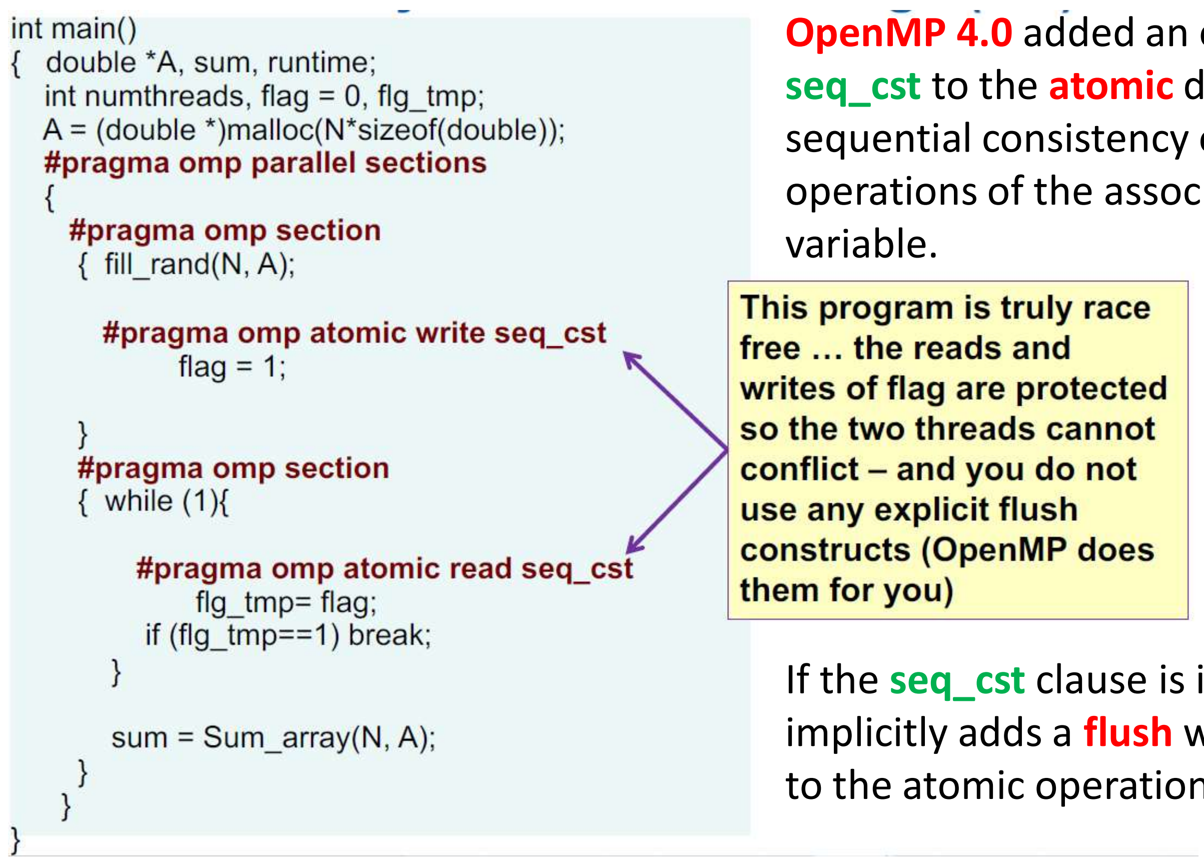
```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
      {  fill_rand(N, A);
         #pragma omp flush
         #pragma omp atomic write
               flag = 1;
         #pragma omp flush (flag)
      }
      #pragma omp section
      {  while (1){
            #pragma omp flush(flag)
            #pragma omp atomic read
               flg_tmp= flag;
            if (flg_tmp==1) break;
         }
         #pragma omp flush
         sum = Sum_array(N, A);
      }
    }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict
Still painful and error prone due to all of the flushes that are required

The problem with the code on the last slide is that the reads ands write of the shared **flag** variable are necessarily atomic operations.
If not protected, a **data race** can occur.

Figure Credit: [3]

OpenMP: Flush



```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
      {  fill_rand(N, A);

         #pragma omp atomic write seq_cst
                flag = 1;

      }
      #pragma omp section
      {  while (1){

            #pragma omp atomic read seq_cst
                  flg_tmp= flag;
              if (flg_tmp==1) break;
          }

          sum = Sum_array(N, A);
      }
    }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict – and you do not use any explicit flush constructs (OpenMP does them for you)

OpenMP 4.0 added an optional clause **seq_cst** to the **atomic** directive to impose sequential consistency on reads and writes operations of the associated atomic variable.

If the **seq_cst** clause is included, OpenMP implicitly adds a **flush** without a **flush-set** to the atomic operation.

Figure Credit: [3]

References

[1] *L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Comput. 28, 9 (September 1979), 690–691. DOI:https://doi.org/10.1109/TC.1979.1675439*

[2] *Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. Computer 29, 12 (December 1996), 66–76. DOI:https://doi.org/10.1109/2.546611*

[3] *Tim Mattson. 2020. A Hands-on Introduction to OpenMP.*