OpenMP[*] Overview:

```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$OM
C$OM
C$O
ED
C
#pr
```

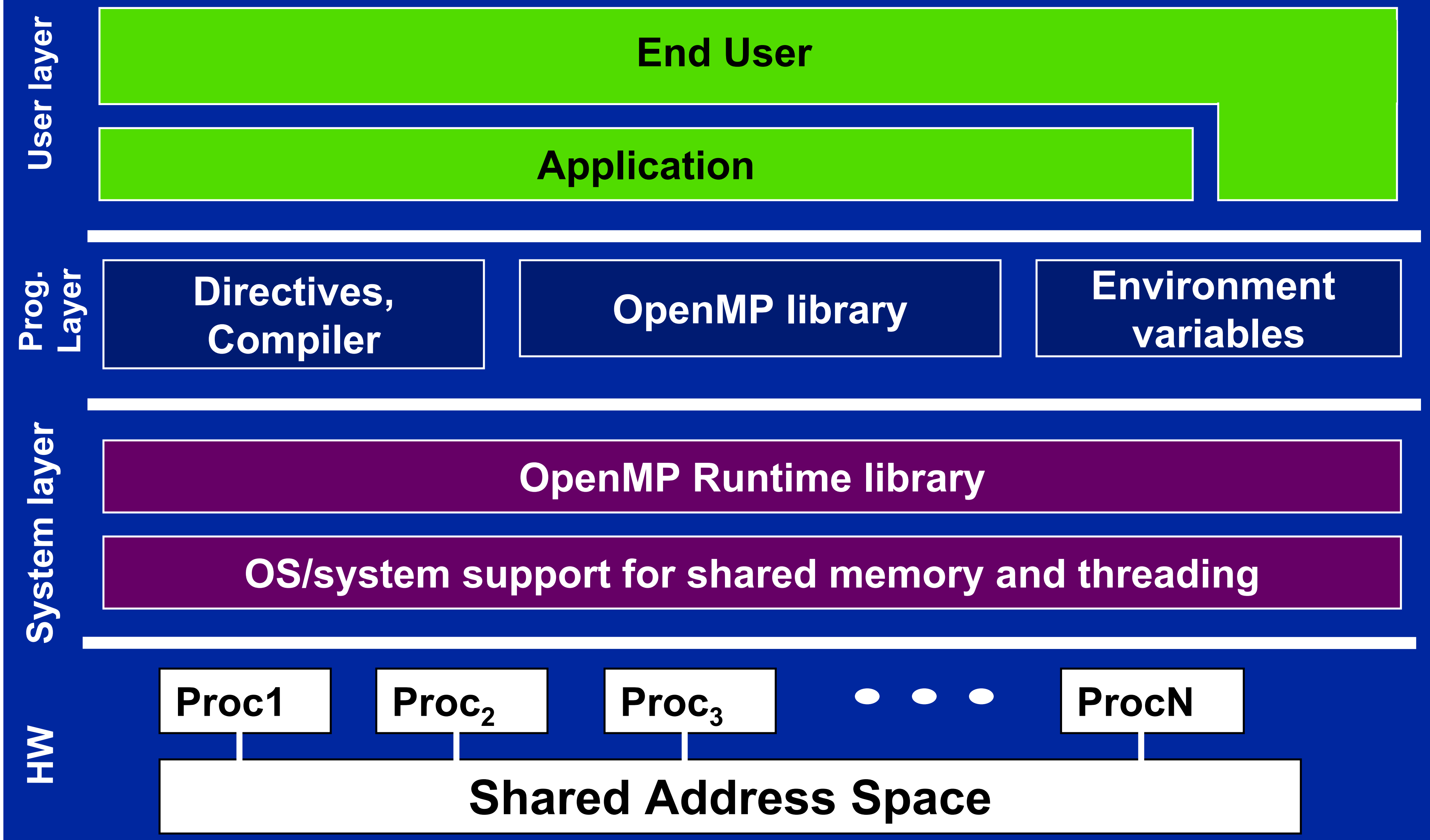
OpenMP: An API for Writing Multithreaded Applications

- **A set of compiler directives and library routines for parallel application programmers**
- **Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++**
- **Standardizes last 20 years of SMP practice**

```
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP Basic Defs: Solution Stack



OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp ***construct [clause [clause]…]***

  - Example

    #pragma omp parallel num_threads(4)

- Function prototypes and types in the file:

  #include <omp.h>

- Most OpenMP* constructs apply to a "structured block".
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It's OK to have an exit() within the structured block.

Exercise 1, Part A: Hello world

Verify that your environment works

- **Write a program that prints “hello world”.**

```
void main()
{



    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```
#include “omp.h”
void main()
{
 #pragma omp parallel
 {

    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
 }
}
```

Switches for compiling and linking

-fopenmp	**gcc**
-mp	**pgi**
/Qopenmp	**intel**

Exercise 1: Solution
A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
void main()
{
#pragma omp parallel
 {
    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

```
hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)
```

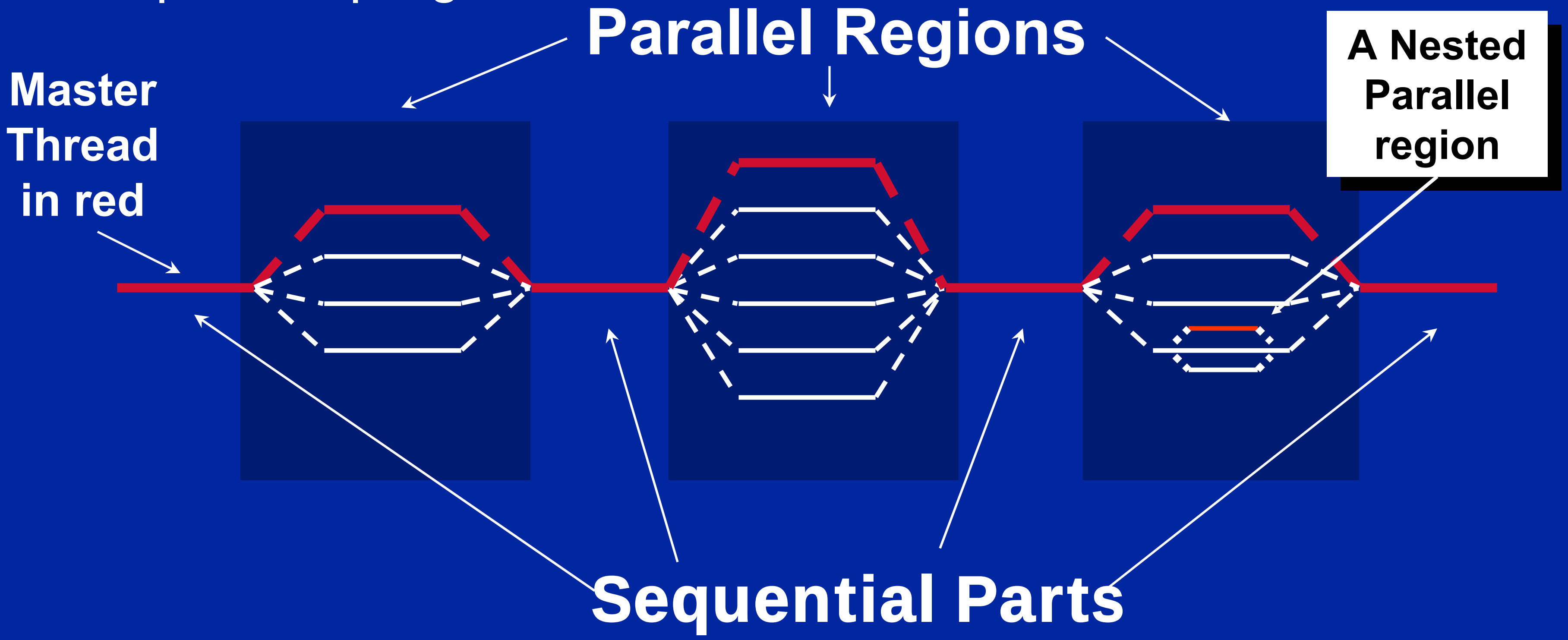
OpenMP Overview:
How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

OpenMP Programming Model:

Fork-Join Parallelism:

- ◆Master thread spawns a team of threads as needed.
- ◆Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for `ID = 0` to `3`

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions

- **You create threads in OpenMP* with the parallel construct.**
- **For example, To create a 4 thread Parallel region:**

```
double A[1000];

#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block

clause to request a certain number of threads

Runtime function returning a thread ID

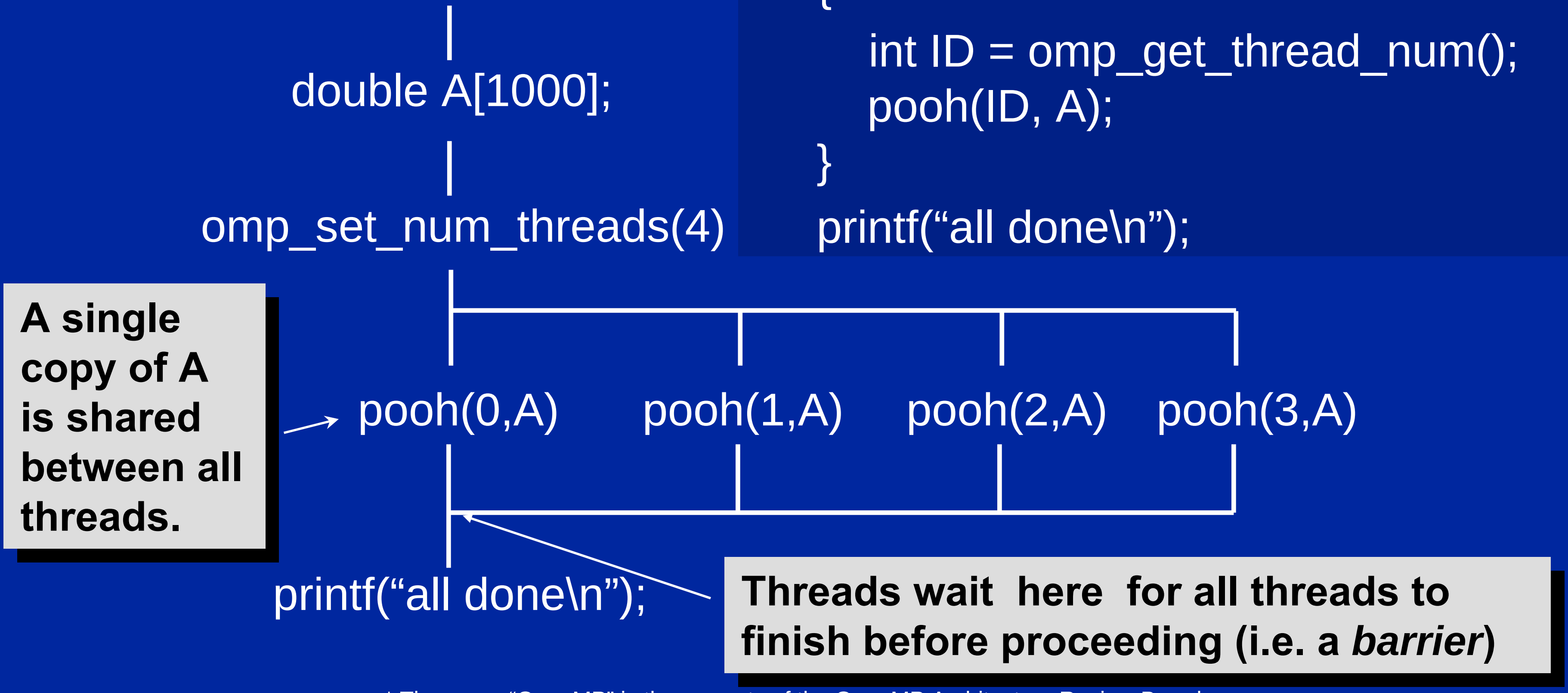
- **Each thread calls `pooh(ID,A)` for `ID = 0` to `3`**

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions example

- **Each thread executes the same code redundantly.**

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

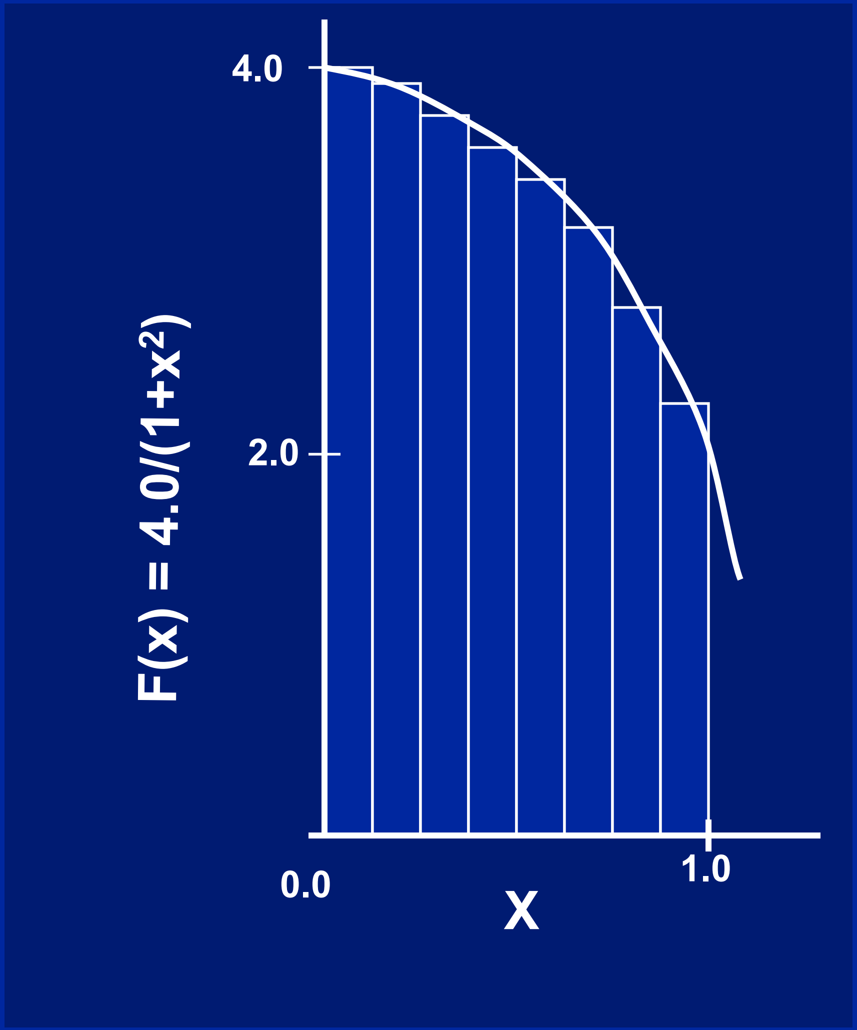


A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

Exercises 2 to 4:
Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Exercises 2 to 4: Serial PI Program

```
static long num_steps = 100000;
double step;
void main ()
{         int i;   double x, pi, sum = 0.0;

          step = 1.0/(double) num_steps;

          for (i=0;i< num_steps; i++){
                   x = (i+0.5)*step;
                   sum = sum + 4.0/(1.0+x*x);
          }
          pi = step * sum;
}
```

Exercise 2

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Synchronization is used to impose order constraints and to protect access to shared data

Discussed later

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```
float res;

#pragma omp parallel

{     float B;   int i, id, nthrds;

      id = omp_get_thread_num();

      nthrds = omp_get_num_threads();

       for(i=id;i<niters;i+nthrds){

             B =  big_job(i);

#pragma omp critical
               consume (B, res);

      }
}
```

Threads wait their turn – only one at a time calls consume()

Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel

{
         double tmp, B;

        B =  DOIT();

        tmp = big_ugly(B);

 #pragma omp atomic
           X +=  tmp;

}
```

Atomic only protects the read/update of X

SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct …. Coming in OpenMP 3.0

Discussed later

The loop worksharing Constructs

- The loop workharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
}
```

Loop construct name:

- **C/C++: for**
- **Fortran: do**

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Combined parallel/worksharing construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```
 double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
```

These are equivalent

Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Note: loop index "i" is private by default

Remove loop carried dependence

```
int i,  A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    int j = 5 + 2*i;
     A[i] = big(j);
}
```

Reduction

- How do we handle this case?

```
  double  ave=0.0, A[MAX];    int i;
  for (i=0;i< MAX; i++) {
      ave + = A[i];
 }
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Compiler finds standard reduction expressions containing “op” and uses them to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
  double  ave=0.0, A[MAX];    int i;
  #pragma omp parallel for reduction (+:ave)
  for (i=0;i< MAX; i++) {
      ave + = A[i];
 }
ave = ave/MAX;
```

Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
        id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Master Construct

- The master construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp master
        {    exchange_boundaries();   }
#pragma omp  barrier
        do_many_other_things();
}
```

Single worksharing Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();
}
```

Synchronization: ordered

- The ordered region executes in the sequential order.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res += consum(tmp);
        }
```

Synchronization: Lock routines

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()

A lock implies a memory fence (a "flush") of all thread visible variables

- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

Synchronization: Simple Locks

- Protect resources with locks.

```
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);
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

Free-up storage when done.

Runtime Library routines

- **Runtime environment routines:**
  - **Modify/Check the number of threads**
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - **Are we in an active parallel region?**
    - omp_in_parallel()
  - **Do you want the system to dynamically vary the number of threads from one parallel construct to another?**
    - omp_set_dynamic,   omp_get_dynamic();
  - **How many processors in the system?**
    - omp_num_procs()

...plus a few less commonly used routines.

Runtime Library routines

- To use a known, fixed number of threads in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

```
#include <omp.h>
void main()
{   int num_threads;
      omp_set_dynamic( 0 );
      omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
   {     int id=omp_get_thread_num();
#pragma omp single
             num_threads = omp_get_num_threads();
          do_lots_of_stuff(id);
    }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

Environment Variables

- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*
- Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.
  - OMP_SCHEDULE "schedule[, chunk_size]"

… Plus several less commonly used environment variables.

Data environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

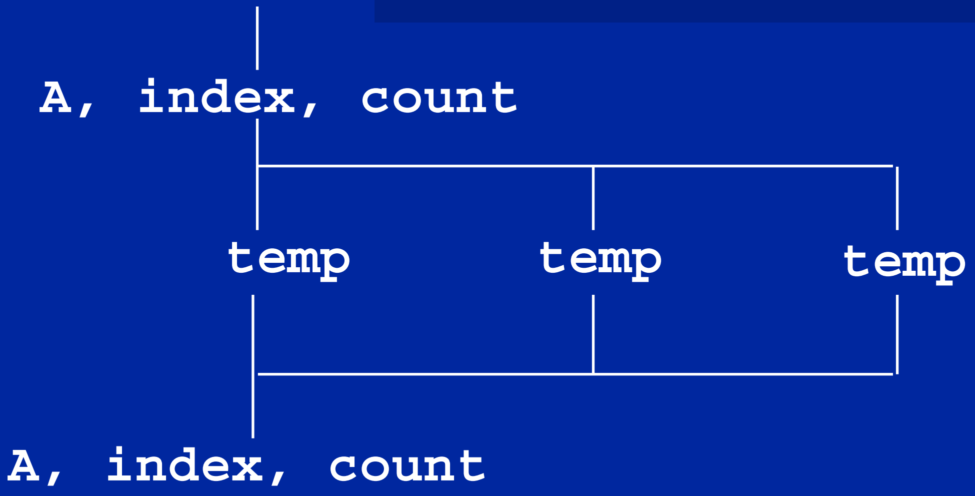
Data sharing: Examples

```
 double A[10];
 int main() {
 int index[10];
#pragma omp parallel
      work(index);
 printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



* Third party trademarks and names are the property of their respective owner.

Data sharing: Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)

    DEFAULT(PRIVATE) *is Fortran only*

All data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.

Data Sharing: Private Clause

- **private(var) creates a new local copy of var for each thread.**
  - **The value is uninitialized**
  - **In OpenMP 2.5 the value of the shared variable is undefined after the region**

```
void wrong() {
     int tmp = 0;
#pragma omp for private(tmp)
     for (int j = 0; j < 1000; ++j)
                tmp += j;
     printf("%d\n", tmp);
}
```

tmp was not initialized

tmp: 0 in 3.0, unspecified in 2.5

Data Sharing: Private Clause When is the original variable valid?

- **The original variable's value is unspecified in OpenMP 2.5.**
- **In OpenMP 3.0, if it is referenced outside of the construct**
  - **Implementations may reference the original variable or a copy ….. A dangerous programming practice!**

```
int tmp;
void danger() {
      tmp = 0;
#pragma omp parallel private(tmp)
      work();
     printf("%d\n", tmp);
}
```

tmp has unspecified value

```
extern int tmp;
void work() {
      tmp = 5;
}
```

unspecified which copy of tmp

Data Sharing: Firstprivate Clause

- **Firstprivate is a special case of private.**
  - **Initializes each private copy with the corresponding value from the master thread.**

```
void useless() {
    int tmp = 0;
#pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j)
      tmp += j;
    printf(“%d\n”, tmp);
 }
```

Each thread gets its own tmp with an initial value of 0

tmp: 0 in 3.0, unspecified in 2.5

Data sharing: Lastprivate Clause

- **Lastprivate passes the value of a private from the last iteration to a global variable.**

```
void closer() {
    int tmp = 0;
#pragma omp parallel for firstprivate(tmp) \
   lastprivate(tmp)
   for (int j = 0; j < 1000; ++j)
      tmp += j;
   printf(“%d\n”, tmp);
}
```

Each thread gets its own tmp with an initial value of 0

tmp is defined as its value at the “last sequential” iteration (i.e., for j=999)

Data Sharing:
A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
          variables A,B, and C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- "A" is shared by all threads; equals 1
- "B" and "C" are local to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

Outside this parallel region ...

- The values of "B" and "C" are unspecified in OpenMP 2.5, and in OpenMP 3.0 if referenced in the region but outside the construct.

Data Sharing: Default Clause

- Note that the default storage attribute is **DEFAULT(SHARED)** (so no need to use it)
  - Exception: #pragma omp task
- To change default: **DEFAULT(PRIVATE)**
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- **DEFAULT(NONE)**: *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

Sections worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{

   #pragma omp sections
   {
   #pragma omp section
            X_calculation();
   #pragma omp section
             y_calculation();
   #pragma omp section
             z_calculation();
   }

}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.

loop worksharing constructs:
The schedule clause

- **The schedule clause affects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).

loop work-sharing constructs:
The schedule clause

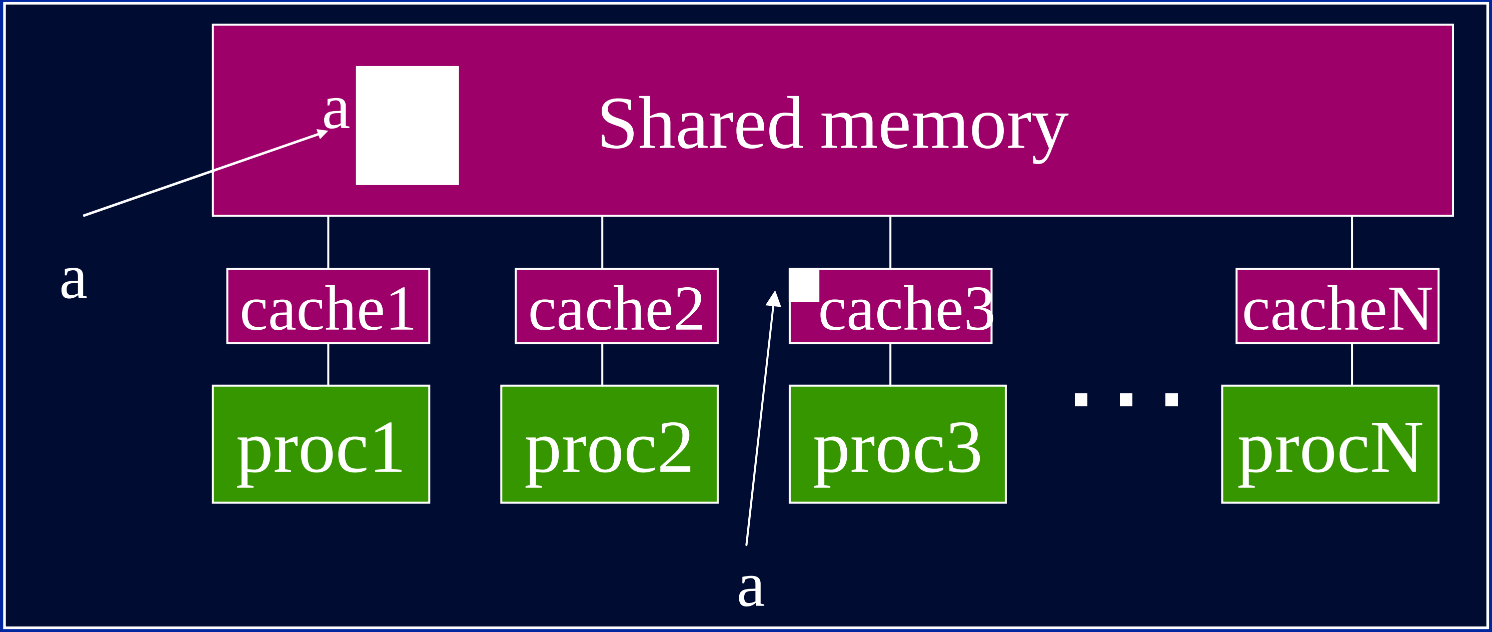
Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

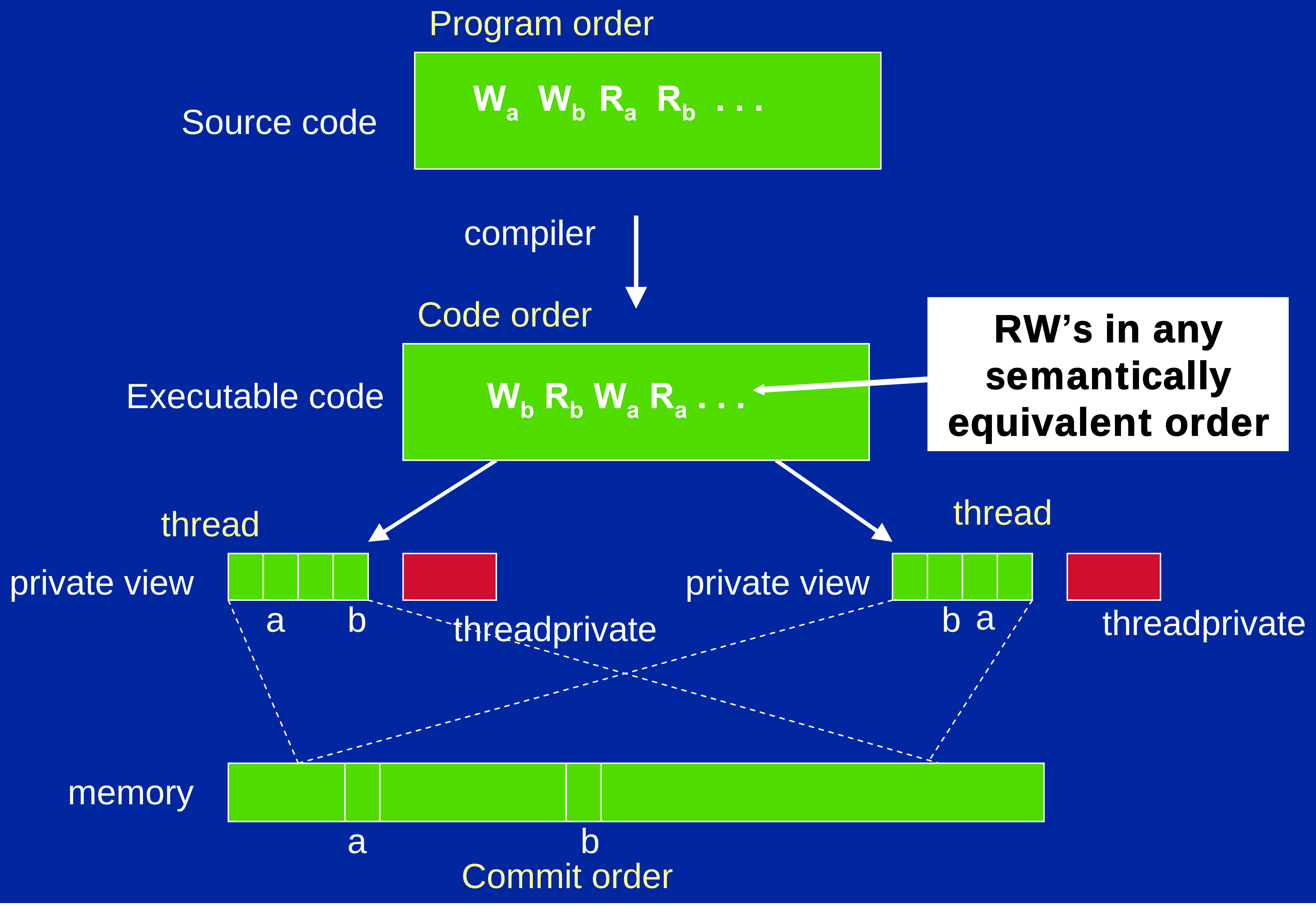
OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:



- A memory model is defined in terms of:
  - **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.

OpenMP Memory Model: Basic Terms



Consistency: Memory Access Re-ordering

- Re-ordering:
  - Compiler re-orders program order to the code order
  - Machine re-orders code order to the memory commit order
- At a given point in time, the temporary view of memory may vary from shared memory.
- Consistency models based on orderings of Reads (R), Writes (W) and Synchronizations (S):
  - $R \rightarrow R$, $W \rightarrow W$, $R \rightarrow W$, $R \rightarrow S$, $S \rightarrow S$, $W \rightarrow S$

Consistency

- Sequential Consistency:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - They remain in program order for each processor.
    - They are seen to be in the same overall order by each of the other processors.
  - Program order = code order = commit order
- Relaxed consistency:
  - Remove some of the ordering constraints for memory ops (R, W, S).

OpenMP and Relaxed Consistency

- OpenMP defines consistency as a variant of <u>weak consistency</u>:
  - S ops must be in sequential order across threads.
  - Can not reorder S ops with R or W ops on the same thread
    - Weak consistency guarantees

      $S \rightarrow W, \quad S \rightarrow R, R \rightarrow S, W \rightarrow S, S \rightarrow S$
- The Synchronization operation relevant to this discussion is flush.

Flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the "flush set".
- The flush set is:
  - "all thread visible variables" for a flush construct without an argument list.
  - a list of variables when the "flush(list)" construct is used.
- The action of Flush is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don't execute until after the flush.
  - Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read, W = write, S = synchronization

Synchronization: flush example

- Flush forces data to be updated in memory so other threads see the most recent value

```
double A;

A = compute();

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

Note: OpenMP’s flush is analogous to a fence in other shared memory API’s.