OpenMP: An API for Writing Portable SMP Application Software

Effective Standards for parallel programming

Thread Libraries
- Win32 API - a low level approach.
- threads.h++ - a high level class library.

Compiler Directives
- OpenMP - portable shared memory parallelism.

Message Passing Libraries
- MPI - www.mpi-softtech.com

High Level Tools
OpenMP: Introduction

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Makes it easy to create multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 15 years of SMP practice

OpenMP: Supporters*

Hardware vendors
- Intel, HP, SGI, IBM, SUN, Compaq, Fujitsu

Software tools vendors
- KAI, PGI, PSR, APR, Absoft

Applications vendors
- ANSYS, Fluent, Oxford Molecular, NAG, DOE
- ASCI, Dash, Livermore Software, and many others

*These names of these vendors were taken from the OpenMP web site (www.openmp.org). We have made no attempts to confirm OpenMP support, verify conformity to the specifications, or measure the degree of OpenMP utilization.
OpenMP: Programming Model

Fork-Join Parallelism:
**Master thread** spawns a **team of threads** as needed.
Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.

Parallel Program

Sequential Program

OpenMP: How is OpenMP typically used?

OpenMP is usually used to parallelize loops:
- Find your most time consuming loops.
- Split them up between threads.

Split-up this loop between multiple threads

```c
void main()
{
    double Res[1000];
    for(int i=0;i<1000;i++)
    {
        do_huge_comp(&Res[i]);
    }
}
```

```c
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++)
    {
        do_huge_comp(&Res[i]);
    }
}
```
OpenMP: How do threads interact?

OpenMP is a shared memory model.
  – Threads communicate by sharing variables.

Unintended sharing of data can lead to 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 stored to minimize the need for synchronization.

OpenMP: Syntax

Most of the constructs in OpenMP are compiler directives or pragmas.

  – For C and C++, the pragmas take the form:
    #pragma omp construct [clause [clause]…]

Since the constructs are directives, an OpenMP program can be compiled by compilers that don’t support OpenMP.
OpenMP: Structured blocks

Most OpenMP constructs apply to structured blocks.

- Structured block: a block of code with one point of entry at the top and one point of exit at the bottom. The only other branches allowed are STOP statements in Fortran and exit() in C/C++.

```
#pragma omp parallel
{
  loop: wrk[id] = garbage[id];
  res[id] = wrk[id] * wrk[id];
  if( conv( res[id] ) )
    goto loop;
  printf("%d",id);
}
```

A structured block

```
#pragma omp parallel
{
  loop: wrk[id] = garbage[id];
  alt:  res[id]=wrk[id] * wrk[id];
    if( conv( res[id] ) )
      goto printit;
    go to loop;
  }
  if ( not_DONE ) goto alt;
prnt: printf("%d",id);
```

Not a structured block

OpenMP: Contents

OpenMP’s constructs fall into 5 categories:
- Parallel Regions
- Worksharing
- Data Environment
- Synchronization
- Runtime functions/environment variables

OpenMP is basically the same between Fortran and C/C++
OpenMP: Parallel Regions

You create threads in OpenMP with the “omp parallel” pragma.
For example, To create a 4 thread Parallel region:

```c
double a[1000];
omp_set_num_threads( 4 );
#pragma omp parallel
{
    int id = omp_thread_num();
    pooh( id, a );
}
```

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

OpenMP: Parallel Regions

Each thread executes the same code redundantly.

```c
double A[1000];
omp_set_num_threads(4)
printf(“all done\n”);
```

A single copy of A is shared between all threads.

```c
pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,A)
```

Threads wait here for all threads to finish before proceeding (i.e. a barrier)
Exercise 1:
A multi-threaded “Hello world” program

Write a multithreaded program where each thread prints a simple message (such as “hello world”).
Use two separate printf statements and include the thread id:

```c
int id = omp_get_thread_num();
printf(" hello(%d) ", id);
printf(" world(%d) ", id);
```

What do the results tell you about I/O with multiple threads?

OpenMP: Some subtle details (don’t worry about these at first)

Dynamic mode (the default mode):
- The number of threads used in a parallel region can vary from one parallel region to another.
- Setting the number of threads only sets the maximum number of threads - you could get less.

Static mode:
- The number of threads is fixed and controlled by the programmer.

OpenMP lets you nest parallel regions, but...
- A compiler can choose to serialize the nested parallel region (i.e. use a team with only one thread).
OpenMP: Contents

OpenMP’s constructs fall into 5 categories:
- Parallel Regions
- Worksharing
- Data Environment
- Synchronization
- Runtime functions/environment variables

OpenMP: Work-Sharing Constructs

The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```c
#pragma omp parallel
#pragma omp for
    for ( i=0; i<n; i++ ){
        doNeatStuff( i );
    }
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.
Work Sharing Constructs
A motivating example

Sequential code

```
for( i=0; i<n; i++ )   { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel
{
    int id, i, n, numThreads, istart, iend;
    id = omp_get_thread_num();
    numThreads = omp_get_num_threads();
    istart = id * n / numThreads;
    iend = (id+1) * n / numThreads;
    for( i=istart; i<iend; i++ )   { a[i] = a[i] + b[i]; }
}
```

OpenMP parallel region and a work-sharing for-construct

```
#pragma omp parallel
#pragma omp for schedule(static)
for( i=0; i<n; i++ )   { a[i] = a[i] + b[i]; }
```

OpenMP For construct:
The schedule clause

The schedule clause effects 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.
OpenMP: Work-Sharing Constructs

The *sections* work-sharing construct gives a different structured block to each thread.

```c
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    x_calculation();
    #pragma omp section
    y_calculation();
    #pragma omp section
    zCalculation();
}
```

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

OpenMP: Combined Parallel Work-Sharing Constructs

A short hand notation that combines the Parallel and work-sharing construct.

```c
#pragma omp parallel for
for ( i=0; i<n; i++ ){
    doNeatStuff( i );
}
```

There’s also a “parallel sections” construct.
Exercise 2:
A multi-threaded “pi” program

On the following slide, you’ll see a sequential program that uses numerical integration to compute an estimate of PI.

Parallelize this program using OpenMP. There are several options (do them all if you have time):
– Do it as an SPMD program using a parallel region only.
– Do it with a work sharing construct.

Remember, you'll need to make sure multiple threads don’t overwrite each other’s variables.

PI Program:
The sequential program

```c
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=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```
OpenMP: Contents

OpenMP’s constructs fall into 5 categories:
- Parallel Regions
- Worksharing
- Data Environment
- Synchronization
- Runtime functions/environment variables

OpenMP: More details: Scope of OpenMP constructs

OpenMP constructs can span multiple source files.

```c
#define omp parallel
{ whoami(); }

void whoami()
{
    int iam;
    iam = omp_get_thread_num()
#pragma omp critical
    { printf("Hello from %d\n"), iam
    }
}
```

This print is done sequentially!
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

But not everything is shared...
  – Stack variables in sub-programs called from parallel regions are **PRIVATE**
  – Automatic variables within a statement block are **PRIVATE**.

```c
static int A[10];
main() {
  int index[10];
  input();
  #pragma omp parallel
  work( index );
  printf("%d\n", index[0]);
}
void work (int * index) {
  real temp[10];
  ...........
```

A and index are shared by all threads.

temp is local to each thread
Data Environment: Changing storage attributes

One can selectively change storage attributes constructs using the following clauses*

- SHARED
- PRIVATE
- FIRSTPRIVATE

The value of a private inside a parallel loop can be transmitted to a global value outside the loop with:

- LASTPRIVATE

The default status can be modified with:

- DEFAULT (PRIVATE | SHARED | NONE)

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

Private Clause

- private(var) creates a local copy of var for each thread.
  - The value is uninitialized
  - Private copy is not storage associated with the original
Firstprivate Clause

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

```c
/* program almost_right*/
main() {
    int j, is = 0;
    #pragma omp parallel for private (is)
    for (j=1; j<=1000; j++)
        is = is + j;
    printf("%d\n", is);
}
```

Each thread gets its own `is` with an initial value of 0

On our installation, `is` is defined with the its initial value 0 at this point.

Lastprivate Clause

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

```c
/* program closer*/
main() {
    int j, is = 0;
    #pragma omp parallel for firstprivate (is) lastprivate (is)
    for (j=1; j<=1000; j++)
        is = is + j;
    printf("%d\n", is);
}
```

Each thread gets its own `is` with an initial value of 0

`is` is defined as its value at the last iteration (i.e. for j=1000)
OpenMP: Another data environment example

Here’s an example of PRIVATE and FIRSTPRIVATE

Assume that variables A, B, and C = 1
#pragma omp private ( B )
#pragma omp firstprivate ( C )

Inside this parallel region ...
“A” is shared by all threads and 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 defined as 1 on our implementation.

OpenMP: Default Clause

Note that the default storage attribute is DEFAULT(SHARED) (so no need to specify)
To change default: DEFAULT(PRIVATE)
- each variable in static extent of the parallel region 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

Only the Fortran API supports default(private).
C/C++ only default(shared) or default(none).
OpenMP: Reduction

- Another clause that effects the way variables are shared:
  - reduction (op : list)
- The variables in “list” must be shared in the enclosing parallel region.
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
  - pair wise “op” is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.

OpenMP: Reduction example

```c
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i;
    double zz, func(), result=0.0;
    omp_set_num_threads(NUM_THREADS)
#pragma omp parallel for reduction(+:result) private(zz)
    for (i=0; i< 1000; i++) {
        zz = func( i );
        result = result + zz;
    }
} 
```
Exercise 3: A multi-threaded “pi” program

Return to your “pi” program and this time, use private, reduction and a worksharing construct to parallelize it. See how similar you can make it to the original sequential program.

OpenMP: Some subtle details (don’t worry about these at first)

The data scope clauses take a list argument
The list can be a common block name with is a short hand for listing all the variables in the common block.

Default private for some loop indices:
Fortran: loop indices are private even if they are specified as shared.
C: Loop indices on “work-shared loops” are private when they otherwise would be shared.

Not all privates are undefined
Allocatable arrays in Fortran
Class type (i.e. non-POD) variables in C++.

See the OpenMP spec. for more details.
OpenMP: More subtle details (don’t worry about these at first)

- Variables privatized in a parallel region can not be reprivilized on an enclosed worksharing directive.
- Assumed size and assumed shape arrays can not be privatized.
- Fortran pointers or allocatable arrays can be private or shared but not lastprivate or firstprivate.
- When a common block is listed in a private, firstprivate, or lastprivate clause, its constituent elements can’t appear in other data scope clauses.
- If an element of a shared common block is privatized, it is no longer storage associated with the common block.

OpenMP: Contents

OpenMP’s constructs fall into 5 categories:
- Parallel Regions
- Worksharing
- Data Environment
- Synchronization
- Runtime functions/environment variables
OpenMP: Synchronization

OpenMP has the following constructs to support synchronization:
- atomic
- critical section
- barrier
- flush
- ordered
- single
- master

We discuss this here, but it really isn't a synchronization construct. It's a work-sharing construct that includes synchronization.

Only one thread at a time can enter a critical section.

```
#pragma omp parallel for private(b) shared(result)
for (i = 1, i <= numIterations, i++) {
    b = doit(i);
    #pragma omp critical
    consume(b, &result);
}
```
OpenMP: Synchronization

Atomic is a special case of a critical section that applies only to the update of a memory location (the update of $x$ in the following example):

```c
#pragma omp parallel private (b)
{
    b = doit ( something );
    #pragma omp atomic
    x= x + b;
}
```

OpenMP: Synchronization

Barrier: Each thread waits until all threads arrive.

```c
#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 );
}
```
OpenMP: Synchronization

The *ordered* construct enforces the sequential order for a block.

```c
#pragma omp parallel private (temp)
#pragma omp for ordered
for ( i=0; i<n; i++ ){
    temp = NEAT_STUFF( i );
    #pragma ordered
    result = consume( temp );
}
```

OpenMP: Synchronization

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

```c
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    exchange_boundaries();
    #pragma barrier
    do_many_other_things();
}
```
OpenMP: Synchronization

The **single** construct denotes a block of code that is executed by only one thread. A barrier and a flush are implied at the end of the single block.

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

OpenMP: Synchronization

The **flush** construct denotes a sequence point where a thread tries to create a consistent view of memory.

- All memory operations (both reads and writes) defined prior to the sequence point must complete.
- All memory operations (both reads and writes) defined after the sequence point must follow the flush.
- Variables in registers or write buffers must be updated in memory.

Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

This is a confusing construct. To learn more, consult the OpenMP specifications.
OpenMP: Implicit synchronization

Barriers are implied at the end of the following OpenMP constructs:

- parallel
- for (except when nowait is used)
- sections (except when nowait is used)
- critical
- single (except when nowait is used)

Flush is implied at the end of the following OpenMP constructs:

- barrier
- critical
- for
- parallel
- sections
- single
- ordered

OpenMP: Some subtle details on directive nesting

- For, sections and single directives binding to the same parallel region can't be nested.
- Critical sections with the same name can't be nested.
- For, sections, and single can not appear in the dynamic extent of critical, ordered or master.
- Barrier can not appear in the dynamic extent of for, ordered, sections, single, master or critical
- Master can not appear in the dynamic extent of for, sections and single.
- Ordered are not allowed inside critical
- Any directives legal inside a parallel region are also legal outside a parallel region in which case they are treated as part of a team of size one.
OpenMP: Contents

OpenMP’s constructs fall into 5 categories:
- Parallel Regions
- Worksharing
- Data Environment
- Synchronization
- Runtime functions/environment variables

OpenMP: Library routines

Lock routines
- `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`

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()`
- Turn on/off nesting and dynamic mode
  - `omp_set_nested()`, `omp_set_dynamic()`, `omp_get_nested()`, `omp_get_dynamic()`
- Are we in a parallel region?
  - `omp_in_parallel()`
- How many processors in the system?
  - `omp_num_procs()`
OpenMP: Library Routines

Protect resources with locks.

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

OpenMP: Library Routines

To fix the number of threads used in a program, first turn off dynamic mode and then set the number of threads.

```c
#include <omp.h>
void main()
{
    omp_set_dynamic(0);
    omp_set_num_threads(4);
#pragma omp parallel
{
    int id=omp_get_thread_num();
    do_lots_of_stuff(id);
}
```
## OpenMP: Environment Variables

Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
- OMP_SCHEDULE “schedule[, chunk_size]”

Set the default number of threads to use.
- OMP_NUM_THREADS int literal

Can the program use a different number of threads in each parallel region?
- OMP_DYNAMIC TRUE || FALSE

Will nested parallel regions create new teams of threads, or will they be serialized?
- OMP_NESTED TRUE || FALSE

---

## SMP Programming errors

Shared memory parallel programming is a mixed bag:
- It saves the programmer from having to map data onto multiple processors. In this sense, its much easier.
- It opens up a range of new errors coming from unanticipated shared resource conflicts.
2 major SMP errors

Race Conditions
– The outcome of a program depends on the detailed timing of the threads in the team.

Deadlock
– Threads lock up waiting on a locked resource that will never become free.

Race Conditions

```c
#pragma omp parallel sections
{
    #pragma omp section
    a = b + c;
    #pragma omp section
    b = a + c;
    #pragma omp section
    c = a + b;
}
```

The result varies unpredictably based on detailed order of execution for each section. Wrong answers produced without warning!
**Race Conditions: A complicated solution**

```c
count = 0;
#pragma omp parallel sections
{
  #pragma omp section
  {
    a = b + c;
    count = 1;
    #pragma omp flush count
  }
  #pragma omp section
  {
    b = a + c;
    count = 2;
    #pragma omp flush count
  }
  #pragma omp section
  {
    c = b + a;
    #pragma omp flush count
  }
}
```

In this example, we choose the assignments to occur in the order a, b, c.
- `count` forces this order.
- `flush` so each thread sees updates to `count`.

NOTE: you need the flush on each read and each write.

---

**Race Conditions**

```c
#pragma omp parallel shared (x) private (temp)
{
  id = omp_get_thread_num();
  #pragma omp for reduction (+:x) nowait
  for ( i = 1; i <= 100; i++ ) {
    temp = firstWork( i );
    x = x + temp;
  }
  y[id] = secondWork( x, id );
}
```

The result varies unpredictably because the value of X isn’t dependable until the barrier at the end of the do loop.
Wrong answers produced without warning!

Solution: Be careful when you use NOWAIT.
Race Conditions

```c
float temp, x;
#pragma omp parallel for reduction(+:x)
for ( i = 1; i <= 100; i++ ) {
    temp = firstWork(i);
    x = x + temp;
}
y[id] = secondWork( x, id );
```

The result varies unpredictably because access to shared variable `temp` is not protected. Wrong answers produced without warning!

The user probably wanted to make `TMP` private.

The KAI tool Assure can identify this type of problem immediately!

Exercise 4:
Race conditions and the “pi” program

Return to your “pi” program and this time, drop the private clause on `x`. In other words, let all threads use the same global variable for `x`.

- Does your program still work?
- Run it many times and see what happens to the answer.
- Change the number of threads. Does the answer change?
Deadlock

This shows a race condition and a deadlock.
If A is locked by one thread and B by another, you have deadlock.
If the same thread gets both locks, you get a race condition - i.e. different behavior depending on detailed interleaving of the thread.

Avoid nesting different locks.

```c
omp_init_lock( &lockA );
omp_init_lock( &lockB );
#pragma omp parallel sections
#pragma omp section
{  omp_set_lock( lockA );
   omp_set_lock( lockB );
   useAandB( &results );
  omp_unset_lock( lockB);
   omp_unset_lock( lockA);
}
#pragma omp section
{  omp_set_lock( lockB );
   omp_set_lock( lockA );
   useAandB( &results );
  omp_unset_lock( lockA);
   omp_unset_lock( lockB);
}
```

Deadlock

This shows a race condition and a deadlock.
If A is locked in the first section and the if statement branches around the unset lock, threads running the other sections deadlock waiting for the lock to be released.

Make sure you release your locks.

```c
omp_init_lock( &lockA );
#pragma omp parallel sections
#pragma omp section
{  omp_set_lock( lockA );
iValue = doWork();
   if ( iValue == tolerance )
     omp_unset_lock( lockA );
   else
     writeError( iValue );
}
#pragma omp section
{  omp_set_lock( lockA );
   useAandB( &results );
   omp_unset_lock( lockA );
}
```
OpenMP death-traps

- Are you using threadsafe libraries?
- I/O inside a parallel region can interleave unpredictably.
- Make sure you understand what your constructors are doing with private objects.
- Private variables can mask globals.
- Understand when shared memory is coherent. When in doubt, use `flush`.
- `nowait` removes implied barriers.

Navigating through the Danger Zones

Option 1: Analyze your code to make sure every semantically permitted interleaving of the threads yields the correct results.
- This can be prohibitively difficult due to the explosion of possible interleavings.
- Tools like KAI's Assure can help.
Navigating through the Danger Zones

Option 2: Write SMP code that is portable and equivalent to the sequential form.
- Use a safe subset of OpenMP.
- Follow a set of “rules” for Sequential Equivalence.

Portable Sequential Equivalence

What is Portable Sequential Equivalence (PSE)?
- A program is sequentially equivalent if its results are the same with one thread and many threads.
- For a program to be portable (i.e. runs the same on different platforms/compilers) it must execute identically when the OpenMP constructs are used or ignored.
Portable Sequential Equivalence

Advantages of PSE
- A PSE program can run on a wide range of hardware and with different compilers - minimizes software development costs.
- A PSE program can be tested and debugged in serial mode with off the shelf tools - even if they don't support OpenMP.

2 Forms of Sequential Equivalence

Two forms of Sequential equivalence based on what you mean by the phrase “equivalent to the single threaded execution”:
- Strong SE: bitwise identical results.
- Weak SE: equivalent mathematically but due to quirks of floating point arithmetic, not bitwise identical.
Strong Sequential Equivalence: rules

- Control data scope with the base language
  - Avoid the data scope clauses.
  - Only use private for scratch variables local to a block (e.g. temporaries or loop control variables) whose global initialization don’t matter.
- Locate all cases where a shared variable can be written by multiple threads.
  - The access to the variable must be protected.
  - If multiple threads combine results into a single value, enforce sequential order.
  - Do not use the reduction clause.

Strong Sequential Equivalence: example

```c
#pragma omp parallel private ( i, temp )
{
  #pragma omp for ordered
  for ( i=1; i<=dimension, i++ ) {
    temp = algebraKernel( i );
    #pragma omp ordered
    combine( temp, &results );
  }
}
```

Everything is shared except \( i \) and \( temp \). These can be private since they are not initialized and they are unused outside the loop. The summation into \( results \) occurs in the sequential order so the result from the program is bitwise compatible with the sequential program.

Problem: Can be inefficient if threads finish in an order that’s greatly different from the sequential order.
Weak Sequential equivalence

- For weak sequential equivalence only mathematically valid constraints are enforced.
  * Floating point arithmetic is not associative and not commutative.
  * In most cases, no particular grouping of floating point operations is mathematically preferred so why take a performance hit by forcing the sequential order?
    - In most cases, if you need a particular grouping of floating point operations, you have a bad algorithm.
- How do you write a program that is portable and satisfies weak sequential equivalence?
  - Follow the same rules as the strong case, but relax sequential ordering constraints.

Weak equivalence: example

```
#pragma omp parallel private ( i, temp )
{
#pragma omp for
  for ( i=1; i<=dimension, i++ ) {
    temp = algebraKernal( i );
    #pragma omp critical
      combine( temp, &results );
  }
}
```

The summation into `results` occurs one thread at a time, but in any order so the result is not bitwise compatible with the sequential program.

Much more efficient, but some users get upset when low order bits vary between program runs.
Sequential Equivalence isn’t a Silver Bullet

This program follows the weak PSE rules, but it’s still wrong.

In this example, `rand()` may not be thread safe. Even if it is, the pseudo-random sequences might overlap thereby throwing off the basic statistics.

```c
#pragma omp parallel private(j, id, temp, rValue)
    id = omp_get_thread_num();
    n = omp_get_num_threads();
    rValue = rand();
#pragma omp for
    for (j = 1, j <= dimension, j++) {
        rValue = rand();
        temp = AlgebraKernel(rValue);
        #pragma omp critical
            combine(temp, &results);
    }
```

Conclusion

OpenMP is:
- A great way to write fast executing code.
- Your gateway to special, painful errors.

You can save yourself grief if you consider the possible danger zones as you write your OpenMP programs.

Tools and/or a discipline of writing portable sequentially equivalent programs can help.