OpenMP Troubleshooting

Data race conditions

One of the biggest drawbacks of shared-memory parallel programming is that it might lead to introduction of a certain type of bug that manifests itself through silent data corruption.

To make matters worse, the runtime behavior of code with this kind of error is also not always reproducible: if one executes the same erroneous program a second time, the problem might not show up.

Data race conditions

```
for (i=0; i<n-1; i++)
   a[i] = a[i+1] + b[i];
```

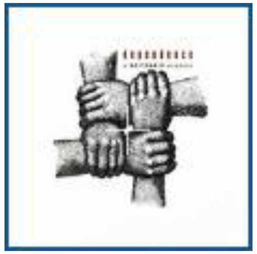
The loop iterations are dependent on each other.
This is called *loop-carried dependence*.

If this loop is carried out in parallel, the result is dependent on the relative speed of the executing threads. This is referred to as a *data race condition*.

```
threads:   1 checksum   1953 correct value   1953
threads:   1 checksum   1953 correct value   1953
threads:   1 checksum   1953 correct value   1953
threads:   1 checksum   1953 correct value   1953

threads:   2 checksum   1953 correct value   1953
threads:   2 checksum   1953 correct value   1953
threads:   2 checksum   1953 correct value   1953
threads:   2 checksum   1953 correct value   1953

threads:   4 checksum   1905 correct value   1953
threads:   4 checksum   1905 correct value   1953
threads:   4 checksum   1953 correct value   1953
threads:   4 checksum   1937 correct value   1953

threads:  32 checksum   1525 correct value   1953
threads:  32 checksum   1473 correct value   1953
threads:  32 checksum   1489 correct value   1953
threads:  32 checksum   1513 correct value   1953

threads:  48 checksum    936 correct value   1953
threads:  48 checksum   1007 correct value   1953
threads:  48 checksum    887 correct value   1953
threads:  48 checksum    822 correct value   1953
```

Figure 7.1: **Output from a loop with a data race condition** - On a single thread the results are always correct, as is to be expected. Even on two threads the results are correct. Using four threads or more, the results are wrong, except in the third run. This demonstrates the non-deterministic behavior of this kind of code.

Data dependence analysis



```
for (i = 0; i < N; i++) {
    a[expression1] = ...;
    ... = a[expression2];
}
```

Necessary condition for this loop to be parallelizable:

`expression1` in any iteration is different from
`expression2` in any other iteration

Default data-sharing attributes

```
int i, j;
#pragma omp parallel for
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        a[i][j] = compute(i,j);
    }
```

Figure 7.5: **Example of a loop variable that is implicitly shared** – Loop variable `i` is private by default, but this is not the case for `j`: it is shared by default. This results in undefined runtime behavior.

Values of private variables

```
void main ()
{
   ..............
   #pragma omp parallel for private(i,a,b)
   for (i=0; i<n; i++)
   {
          b++;
          a = b+i;
   } /*-- End of parallel for --*/
   c = a + b;
   ..............
}
```

Figure 7.6: **Incorrect use of the private clause** – This code has two problems. First, variable `b` is used but not initialized within the parallel loop. Second, variables `a` and `b` should not be used after the parallel loop. The values after the parallel loop are undefined and therefore implementation dependent.

Cont'd on next page

Values of private variables

```
void main ()
{
   .............
   #pragma omp parallel for private(i), firstprivate(b) \
           lastprivate(a,b)
   for (i=0; i<n; i++)
   {
       b++;
       a = b+i;
   } /*-- End of parallel for --*/
   c = a + b;
   .............
}
```

Figure 7.7: **Corrected version using `firstprivate` and `lastprivate` variables** – This is the correct version of the code in Figure 7.6.

Problems with the master construct

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   int Xinit, Xlocal;

   #pragma omp parallel shared(Xinit) private(Xlocal)
   {
     #pragma omp master
     {Xinit = 10;}

     Xlocal = Xinit; /*-- Xinit might not be available yet --*/

   } /*-- End of parallel region --*/
}
```

Figure 7.8: **Incorrect use of the master construct** – This code fragment implicitly assumes that variable `Xinit` is available to the threads after initialization. This is incorrect. The master thread might not have executed the assignment when another thread reaches it, or the variable might not have been flushed to memory.

Assumptions on work scheduling

```
#pragma omp parallel
{
    #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
            z[i] = sqrt(b[i]);
}
```

Figure 7.9: **Example of incorrect assumptions about work scheduling in the OpenMP 2.5 specifications** – The `nowait` clause might potentially introduce a data race condition, even with static work scheduling, if `n` is not a multiple of the number of threads.

Invalid nesting of directives

```
#pragma omp parallel shared(n,a,b)
  {
    #pragma omp for
    for (int i=0; i<n; i++)
    {
        a[i] = i + 1;
        #pragma omp for // WRONG - Needs a new parallel region
        for (int j=0; j<n; j++)
            b[i][j] = a[i];
    }
  } /*-- End of parallel region --*/
```

Figure 7.10: **Example of incorrectly nested directives** – Nested parallelism is implemented at the level of parallel regions, not work-sharing constructs, as erroneously attempted in this code fragment.

Cont'd on next page

Invalid nesting of directives

```
#pragma omp parallel shared(n,a,b)
{
  #pragma omp for
  for (int i=0; i<n; i++)
  {
      a[i] = i + 1;
      #pragma omp parallel for // Okay - This is a parallel region
      for (int j=0; j<n; j++)
          b[i][j] = a[i];
  }
} /*-- End of parallel region --*/
```

Figure 7.11: **Example of correctly nested directives** – This is correct use of nested parallelism. This code fragment has two nested parallel regions.

Subtle errors in the use of directives

```
#pragma omp parallel // Incorrect use of the barrier
{
  if ( omp_get_thread_num() == 0 )
  {
        .....
     #pragma omp barrier
  }
  else
  {
        .....
     #pragma omp barrier
  }
} /*-- End of parallel region --*/
```

Figure 7.12: **Illegal use of the barrier** – The barrier is not encountered by all threads in the team, and therefore this is an illegal OpenMP program. The runtime behavior is undefined.

Subtle errors in the use of directives

```
main()
{
#pragma omp parallel
   {
      work1(); /*-- Executed in parallel --*/
      work2(); /*-- Executed in parallel --*/
   }

#pragma omp parallel
   work1(); /*-- Executed in parallel  --*/
   work2(); /*-- Executed sequentially --*/
}
```

Figure 7.14: **Example of the impact of curly brackets on parallel execution** – It is very likely an error was made in the definition of the second parallel region: function `work2` is executed by the master thread only.

Verification of the sequential version

- Run the source code through syntax checking tools such as *splint* ~~*lint or ftncheck*~~.
- Enable as many compiler diagnostic options as possible. *-Wall*
- Try different compiler optimizations. The bug might already show up for a specific set of options applied to the sequential version.
- Run the loops parallelized with OpenMP backwards. If the result is wrong, the loop(s) cannot be executed in parallel. The reverse is not true. If the result is okay, it does not automatically mean the loop can be parallelized.

Verification of the parallel version

- Run the OpenMP version of the program on one thread. If the error shows up then, there is most likely a basic error in the code.
- Selectively enable/disable OpenMP directives to zoom in on the part of the program where the error originates.
- If a data race is suspected:
  - Use as many threads as possible. The higher the number of threads, the more likely the data race is to show up.
  - ~~DATA and SAVE statements in Fortran programs and~~ the use of `static` and external variables in C/C++ might cause data to be shared unintentionally.
- Check that the libraries used are thread-safe in case one or more of their functions are called within a parallel region.

Debugging tools

The GNU debugger, gdb, allows you to see what is going on 'inside' a program while it executes.

```
gdb program
```

Compile the program with `-g` option in order to produce debugging information.

The Intel thread checker, tcheck, allows to find threading errors like data races and deadlocks.

```
tcheck_cl program
```

Program with a data race

prg.c:

```
int main () {
   int count = 0;
   #pragma omp parallel
   { count++; }
   printf("count = %d\n", count);
   return 0;
}
```

```
icc -o prg -openmp -Wall -g prg.c
```

Using the Intel thread checker

`tcheck_cl prg`

First few lines of the report:

ID	Short Description	Severity Name	Count	Context[Best]	Description	1st Access[Best]	2nd Access[Best]
1	Write -> Read data-race	Error	15	"prg.c":5	Memory read at "prg.c":6 conflicts with a prior memory write at "prg.c":6 (flow dependence)	"prg.c":6	"prg.c":6

Using the Valgrind thread checker

```
valgrind --tool=helgrind prg
```

Some lines of the report:

```
==15991== Possible data race during write of size 8 at 0x421E508
==15991==    at 0x400945B: _dl_lookup_symbol_x (dl-lookup.c:321)
==15991==    by 0x400D3B8: _dl_fixup (dl-runtime.c:108)
==15991==    by 0x40132A1: _dl_runtime_resolve (dl-trampoline.S:43)
==15991==    by 0x5040035: start_thread (pthread_create.c:277)
==15991==    by 0x532811C: clone (clone.S:112)
==15991==   Old state: shared-readonly by threads #1, #2
==15991==   New state: shared-modified by threads #1, #2
==15991==   Reason:    this thread, #2, holds no consistent locks
==15991==   Location 0x421E508 has never been protected by any lock
```

32 OpenMP Traps For C++ Developers

Alexey Kolosov
OOO "Program Verification Systems"
Evgeniy Ryzhkov
OOO "Program Verification Systems"
Andrey Karpov
OOO "Program Verification Systems"

http://www.viva64.com/content/articles/parallel-programming/?f=32_OpenMP_traps.html&lang=en&content=parallel-programming

Logical errors

1. Missing openmp compiler option

   You should enable the option at the moment you create your project.

2. Missing parallel keyword
3. Missing omp keyword
4. Missing for keyword

   You should be accurate about the syntax of the directives you use.

5. Unnecessary parallelization

   You should be accurate about the syntax of the directives you use and understand their meaning.

6. Incorrect usage of the ordered clause

   It is necessary to watch over the syntax of the directives you use.

7. Redefining the number of threads in a parallel section

   The number of threads cannot be changed in a parallel section.

8. Using a lock variable without initializing the variable

   A lock variable must be initialized via the omp_init_lock function call.

9. Unsetting a lock from another thread
10. Using a lock as a barrier

    If a thread uses locks, both the lock (omp_set_lock, omp_test_lock) and unlock (omp_unset_lock) functions must be called by this thread.

11. Threads number dependency

    Your code's behavior must not depend on the number of threads which execute the code.

12. Incorrect usage of dynamic threads creation

    If you really need to make your code's behavior depend on the number of threads, you must make sure that the code will be executed by the needed number of threads (dynamic threads creation must be disabled). We do not recommend using dynamic threads creation.

13. Concurrent usage of a shared resource

    Concurrent shared resource access must be protected by a critical section or a lock.

14. Shared memory access unprotected

    Concurrent shared memory access must be protected as an atomic operation (the most preferable option), critical section or a lock.

15. Using the flush directive with a reference type

    Applying the flush directive to a pointer is meaningless since only the variable's value (a memory address, not the addressed memory) is synchronized in this case.

16. Missing flush directive

    Missing flush directive may cause incorrect memory read/write operations.

17. Missing synchronization

    Missing synchronization may also cause incorrect memory read/write operations.

18. An external variable is specified as threadprivate not in all units
19. Uninitialized local variables
20. Forgotten threadprivate directive
21. Forgotten private clause
22. Careless usage of the lastprivate clause
23. Unexpected values of threadprivate variables in the beginning of parallel sections

    We recommend that you do not use the threadprivate directive and the private, firstprivate, lastprivate clauses. We recommend that you declare local variables in parallel sections and perform first/last assignment operations (if they are necessary) with a shared variable.

24. Incorrect worksharing with private variables

    If you parallelize a code fragment which works with private variables using the threads in which the variables were created different threads will get different values of the variables.

25. Some restrictions of private variables

Private variables must not have reference type, since it will cause concurrent shared memory access. Although the variables will be private, the variables will still address the same memory fragment. Class instances declared as private must have explicit copy constructor, since an instance containing references will be copied incorrectly otherwise.

26. Private variables are not marked as such

You must control access modes of your variables. We recommend that developers who are new to OpenMP use the default(none) clause so that they will have to specify access modes explicitly. In particular, loop variables must always be declared as private or local variables.

27. Parallel array processing without iteration ordering

If an iteration execution depends on the result of a previous iteration, you must use the ordered directive to enable iterations ordering.

Performance errors

28. Unnecessary flush directive

    There is no need to use the flush directive in the cases when the directive is implied.

29. Using critical sections or locks instead of the atomic directive

    We recommend that you use the atomic directive to protect elementary operations when it is possible, since using locks or critical sections slows down you program's execution.

30. Unnecessary concurrent memory writing protection

    There is no need protect private or local variables. Also, there is no need to protect a code fragment which is executed by a single thread only.

31. Too much work in a critical section

    Critical sections should contain as little work as possible. You should not put a code fragment which does not work with shared memory into a critical section. Also we do not recommend putting a complex function calls into a critical section.

32. Too many entries to critical sections

We recommend that you decrease the number of entries to and exits from critical sections. For example, if a critical section contains a conditional statement, you can place the statement before the critical section so that the critical section is entered only if the condition is true.