PARTNERSHIP FOR ADVANCED COMPUTING IN EUROPE

OpenMP- How to Write Correct OpenMP Programs

Matic Brank

LECAD laboratory, FS UL

Outline

- Issues
- Initialisation of private variables
- Race conditions
- Synchronisation constructs
- Flush
- Locks
- Barriers
- Summary

Issues

- Correctness is dependent on private/shared variables
- Private: how are they initialised?
- Shared: do we create any race conditions?
- Are the threads really independent?
- Dependency: are we relying on a specific order in which the threads execute the code?

Initialisation of private variables

- Always intialise (all variables).
- Can initialise in the parallel region.
- Can use firstprivate
- Can use a shared/global
- Can use threadprivate variable.

C/C++:

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

Fortran:

```
Integer i
!$omp parallel do private(i)
do i = 1,n
…
end do
```

C/C++:

```
Int limit pi=1000;
#pragma omp threadprivate(limit)
int main(void) {
#pragma omp parallel copyin(limit)
…
}
```

Fortran:

```
Integer(kind=4) :: limit limit=1000
!$omp threadprivate(limit)
!Defined in a module
program test
!$omp parallel copyin(limit)
…
!$end parallel
```

Race condition

- Threads communicate through shared variables. Uncoordinated access of these variables can lead to undesired effects.
  - two threads update (write) a shared variable in the same step of execution, the result is dependent on the way this variable is accessed. This is called a race condition.
  - Suppose that one processor has an updated result in private cache. Second processor wants to access that memory location - but a read from memory will get the old value since original data not yet written back.

Working example

C/C++:

```
int main(void) {
int turn=0;
#pragma omp parallel shared(turn)
{
int tid = omp_get_thread_num();
while (turn != tid)
  ;
printf("This is thread %d\n",tid);
turn = tid+1;
}}
```

Fortran:

```
program test
integer (kind=4) :: turn, tid
turn=0
!$omp parallel shared(turn) private(tid)
tid = omp_get_thread_num()
do while (turn .NE. tid)
end do
write(6,*) 'This is thread ',tid
turn = tid+1
!$omp end parallel
end program test
```

Non working example

C/C++:

```
int main(void) {
int turn=0;
#pragma omp parallel shared(turn)
{
int tid = omp_get_thread_num();
while (turn > 0)
  ;
turn = 1
print(“This is thread %d\n”,tid);
turn = 0;
}}
```

Fortran:

```
program test
integer (kind=4) :: turn, tid
turn=0
!$omp parallel shared(turn) private(tid)
tid = omp_get_thread_num()
do while (turn .GT. 0)
end do
turn = 1
write(6,*) ‘This is thread ‘,tid
turn = 0
!$omp end parallel
end program test
```

Protecting from race condition

- We must protect any shared variables against a race condition
- We can do this using synchronisation constructs
- They allow only one thread at a time access to the code block which modifies a shared variable.

Synchronisation constructs

- Synchronization imposes order constraints and is used to protect access to shared data
- High-Level Synchronisation Constructs:
  - master
  - critical
  - atomic
  - ordered
  - barrier
- Low-Level Synchronisation Constructs:
  - flush
  - locks

Synchronisation constructs

- Critical directive
  - Only one thread at a time can enter a critical section

```
#pragma omp critical [name]
{
...
}
```

```
!$omp critical [name]
  ...
!$omp end critical
```

- Barrier directive
  - Wait all threads to arrive

C/C++:

```
#pragma omp barrier
```

Fortran:

```
!$omp barrier
```

Synchronisation constructs

- Ordered directive
  - Iterations of the enclosed loop will be executed in the same order as if they were executed sequentially

C/C++:

```
#pragma omp for ordered
....
#pragma omp ordered
...
```

Fortran:

```
!$omp do ordered
...
!$omp ordered
  ...
!$omp end ordered
!$omp end do
```

```
#pragma omp parallel for ordered
for(i=0;i<N;i++) {
   #pragma ordered
   printf(" %d\n", i);
}
```

The output will always be 0,1,2,…,N

Flushing

- Each thread has a temporary view of the shared variables.
- Flushing (without a list) synchronised these across all threads.
- There are implicit flushes at the beginning and end of parallel, ordered and critical regions.
- At the end of worksharing (unless nowait)
- At a barrier
- Atomic regions only flush the variables in the region but can perform a full flush
- However there is no implicit flush at the beginning of worksharing constructs and master

Flush directive

C/C++:

```
int main(void) {
int turn=0;
#pragma omp parallel shared(turn)
{
int tid = omp_get_thread_num();
while (turn != tid) {
#pragma omp flush(turn)
  ;
}
printf("This is thread %d\n",tid);
turn = tid+1;
#pragma omp flush(turn)
}}
```

Fortran:

```
program test
integer (kind=4) :: turn, tid
turn=0
!$omp parallel shared(turn) private(tid)
tid = omp_get_thread_num()
do while (turn .NE. tid)
!$omp flush(turn)
end do
write(6,*) 'This is thread ',tid
turn = tid+1
!$omp flush(turn)
!$omp end parallel
end program test
```

Use critical

C/C++:

```
int main(void) {
int turn=0;
#pragma omp parallel shared(turn)
{
int tid = omp_get_thread_num();
while (turn <= tid) {
#pragma omp critical
  if (turn == tid) {
    printf(“This is thread %d\n”,tid);
    turn = tid+1;
  }
}
}}
```

Fortran:

```
program test
integer (kind=4) :: turn, tid
turn=0
!$omp parallel shared(turn) private(tid)
tid = omp_get_thread_num()
do while (turn .LE. tid)
!$omp critical
  if (turn .EQ. tid) then
    write(6,*) ‘This is thread ‘,tid
    turn = tid+1
endif
!$omp end critical
end do
!$omp end parallel
end program test
```

Locks

- Occasionally we may require more flexibility than is provided by critical and atomic directions.
- A lock is a special variable that may be set by a thread. No other thread may set the lock until the thread which set the lock has unset it.
- A lock must be initialised before it is used, and may be destroyed when it is not longer required.
- Lock variables should not be used for any other purpose.

Locks - example

```
for (i=0; i<nvert; i++)
    omp_init_lock(&lock[i]);

#pragma omp parallel for
  for (i=0; j<nedges; j++) {
     omp_set_lock(&lock[edge[j].vert1]);
     degree[edge[j].vert1]++;
     omp_unset_lock(&lock[edge[j].vert1) ;
     omp_set_lock(&lock[edge[j].vert2]);
     degree[edge[j].vert2]++;
     omp_unset_lock(&lock[edge[j].vert2) ;
  }
for (i=0; i<nvert; i++)
   omp_destroy_lock(&lock[i]);
```

```
do i=1,nvert
  call omp_init_lock(ilock)
end do
!$omp parallel do
 do j=1,nedges
   call omp_set_lock(ilock(j)%vert1)
   degree(edge(j)%vert1) ++
   call omp_unset_lock(ilock(j)%vert1)
   call omp_set_lock(ilock(j)%vert2)
   degree(edge(j)%vert2) ++
   call omp_unset_lock(ilock(j)%vert2)
 end do
!$omp end parallel do
do I = 1,nvert
  call omp_destroy_lock(ilock(i))
end do
```

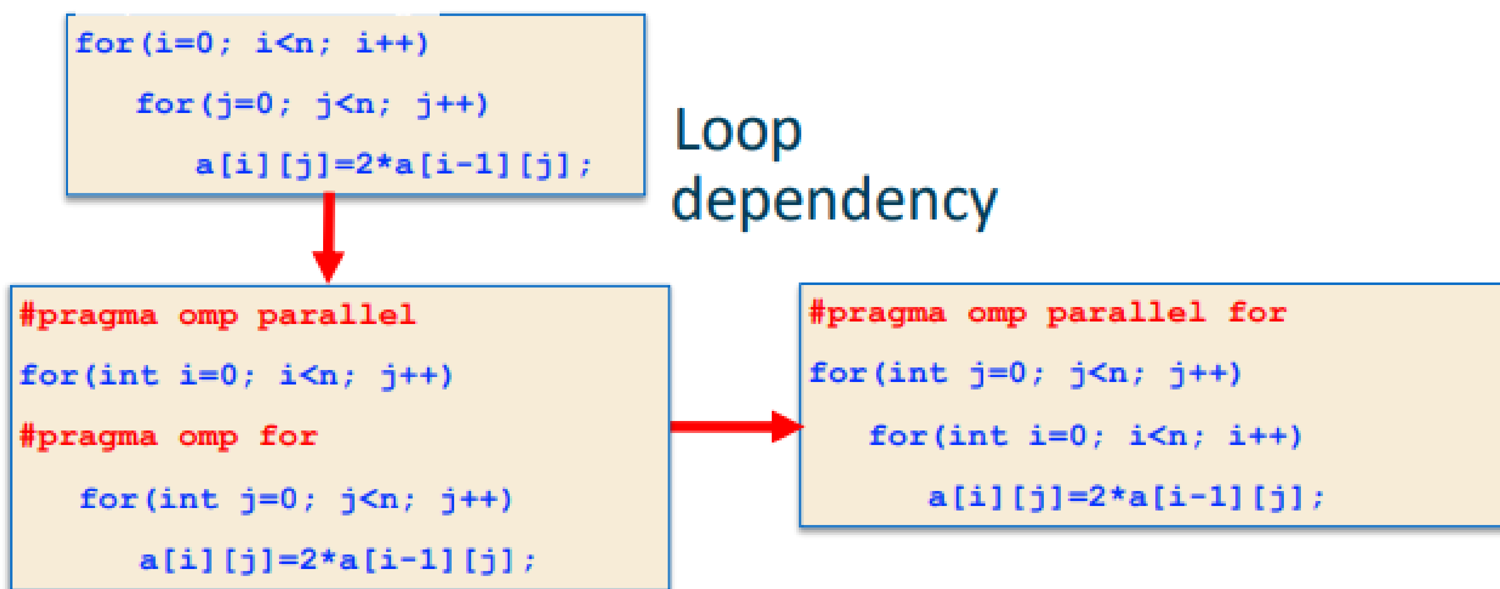
- Vertex degree is the number of edges from that vertex. An edge will join two vertices (vert1 and vert2)

Functions

- Library functions can have hidden race conditions.
- It should say if the function is thread safe or not.
- If it does not say then you can assume that it is not thread safe.
- If thread safe you can use it. • If not then you need to use an alternative.

Independent threads

- Data on one thread can be dependent on data on another thread.
- This can result in wrong answers.
- Thread 1 may require a variable that is calculated on thread 0 – answer



Controlling thread execution

```c
#pragma omp parallel share(a,q)
{
#pragma omp for reduction(+:s) nowait
  for (int i=0; j<n; j++)
     s += a[i];

  int tid;
  tid = omp_get_thread_num();
  q[tid] = f(s,tid);
}
```

```fortran
!$omp parallel share(a,q) private(i,tid)
!$omp do reduction(+:s) nowait
  do i=1,n
    s = s + a(i)
  end do
!$omp end do
 tid = omp_get_thread_num()
 q(tid) = f(s,tid)
!$omp end parallel
```

Summary

- Initialize private variables.
- Protect modification of shared variables.
- Threads work independently or that dependency is maintained.
- Data dependencies are satisfied.
  - Avoid serialisation.
  - Avoid false caching.
  - Avoid too many overheads

THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu