Data Dependency and other Pitfalls

i.e. Programmer Responsibilities

Many aspects of programming, some which are normally handled at a compiler level, have been put on the shoulders of the programmer to figure out.

- Race conditions
- Data dependency
- Thread overheads
- Load imbalance

Each one of these *issues* requires the programmer to think ahead.

Data Dependency

If two statements refer to the same bit of data, there is a *data dependency* between the two statements

```
int main()
{
   int a, b, x = 1;
   a = x;
   b = x + a;
}
```

Data Dependency

Why is this a problem?

```
int main()
{
   int a=1, b=1, x = 100;
   a = x;
   b = a + x;
}
```

Because these data dependant statements limit the parallelism, because changing the *program order* can drastically changes the results of the code

Data Dependency

Statement Ordering

- When a loop is parallelized, the iterations are no longer executed in their program order, so we have to check for dependencies.

- The introduction of tasks also means that parts of a program can be executed in a different order from than they appear in sequential execution.

Data Dependency

Cam loop iterations be executed independently?

- if a data item is read in two different iterations

  We're in the clear!

But,

- if the same item is read in one iteration and written in another
- if the same item is written in two different iterations

  We need a plan

Data Dependency

```
…
#prahma omp parallel for
for (i=2; i < 10; i++)
{
   factorial[i] = i * factorial[i-1];
}
…
```

The compiler will thread this loop, but it will "fail" because at least one of the loop iterations is data-dependent upon a different iteration.

Data Dependency

The three types of dependencies are:

- flow dependencies, or ‘read-after-write’
- anti dependencies, or ‘write-after-read’
- output dependencies, or ‘write-after-write’.

Flow Dependency

```
A = 3

B = A

C = B
```

Anti Dependency

```
A = 2

B = A + 1

A = 5
```

Output Dependency

```
A = 2

B = A + 1

A = 5
```

Flow Dependency

If the read and write occur in same iteration, it is safe to parallelize.

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

However, if the read in a later iteration… there is no *simple* way to parallelize.

```
for (i=0; i<N; i++) {
        .... = ... x[i] ... ;
        x[i+1] = .... ;
}
```

Anti Dependency

The simplest example of anti dependency (write after read) is a reduction

```
for (i=0; i<N; i++) {
          t = t + .....;
}
```

Though, write after reads can be more complex:

```
for (i=0; i<N; i++) {
         x[i] = … x[i+1] … ;
}
```

Anti Dependency (cont)

With a complex write-after-reads

```
for (i=0; i<N; i++) {
        x[i] = … x[i+1] … ;
}
```

We need a bit of strategery

```
for (i=0; i<N; i++)
    xtmp[i] = x[i];
for (i=0; i<N; i++) {
    x[i] = ... xtmp[i+1] ... ;
}
```

Output Dependency

write-after-writes, a little thought:

```
for (i=0; i<N; i++) {
     s[i] = … … ;
     s[i] = s[i] + … … … ;
}
```

Something like that can easily be combined, thereby removing the dependency.

```
for (i=0; i<N; i++) {
     s[i] = … … + … … … ;
}
```

Output Dependency

write-after-writes, a little thought:

```
for (i=0; i<N; i++) {
        t = f(i)
        s += t*t;
}
```

Output Dependency

write-after-writes, a little thought:

```
for (i=0; i<N; i++) {
          t = f(i)
          s += t*t;
}
```

`t` just being a temporary variable, let us declare it private to the thread, thereby leaving an anti dependency which we resolve with simple reduction

Other Pitfalls, things to keep in mind.

i.e. Programmer Responsibilities

- Thread Overhead
  - OpenMP incurs a one-time hit to create its pool of threads at the beginning of your code
    - Worksharing constructs, even though the *act* as if they're creating a new team of threads, are likely pulling from a pool of dormant threads, thread overhead is not icurred again
- Load Balancing
  - Keep in mind, that there is an implicit barrier at the tail of a worksharing construct.
    - unbalanced load distribution will kill your parallel efficiency
    - dynamic *over* static?
    - dynamic and guided can still prove costly, if a particularly *long* job is among the last to be scheduled

Other Pitfalls, things to keep in mind.

i.e. Programmer Responsibilities

- Synchronization
  - Some synchronization constructs are controlled through the operating system calls
    - usually come at a high penalty
    - i.e. critical sections > Amdahl's cost of the loss of parallelism

Other Pitfalls, things to keep in mind.

i.e. Programmer Responsibilities

1. Missing /openmp
2. Missing parallel
3. Missing omp
4. Missing for
5. Unnecessary parallelization
6. Incorrect usage of ordered
7. Redefining the number of threads in a parallel section
8. Using a lock variable without initializing the variable
9. Unsetting a lock from another thread
10. Using lock as a barrier
11. Threads number dependency
12. Incorrect usage of dynamic threads creation
13. Concurrent usage of a shared resource
14. Shared memory access unprotected
15. Using flush with a reference type
16. Missing flush
17. Missing synchronization
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. Incorrect worksharing with private variables
23. Careless usage of the lastprivate clause
24. Unexpected values of threadprivate variables in the beginning of parallel sections
25. Some restrictions of private variables
26. Private variables are not marked as such
27. Parallel array processing without iteration ordering

Performance errors

28. Unnecessary flush
29. Using critical sections or locks instead of the atomic directive
30. Unnecessary concurrent memory writing protection
31. Too much work in a critical section
32. Too many entries to critical sections

https://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers