Introduction to OpenMP

More Syntax and SIMD

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

3.1 SIMD Directives etc.

3.1.1 C/C++ Parallel `for`-loop

C/C++ programmers need to know the rules more precisely. The syntax is:

`for` ([*type*] *var* = *expr* ;
var relop expr ;
increment_expression)

The *increment_expression* can be any of:

var++, ++*var*, *var* += *expr*, *var* = *var* +*expr*,
var−−, −−*var*, *var* −= *expr*, *var* = *var* −*expr*

The constraints are more like Fortran than C or even C++:

- *var* must be a **signed** integer variable.
- *relop* is one of the relational operators.
- Each *expr* must be invariant over the loop; do not include **any** side effects in them.

I recommend using only really simple expressions and, if in doubt, assign expressions to temporary variables outside the loop and use those in the loop control.

3.1.2 More on Clauses

You can specify the scheduling for each loop; you can use it on the `DO`/`for`-loop directives (both combined and split) and only on those. This is the OpenMP scheduling policy, and **not** the system thread scheduling policy done by the kernel.

For normal SIMD work, use `schedule(static)`; specifying it explicitly means that the compiler knows it is using it, and this might generate better code than if it has to look at an environment variable. This divides the loop into equal chunks, and then hands each chunk to a single thread, in turn. Other `schedule` options are described later.

Data environment clauses are allowed on most parallel or work-sharing constructs, and most have the syntax *keyword*(*list*), where *list* is a list of variable names. Most, including `shared` and `private`, can be repeated, though you must not repeat any variable name, of course. For example:

```
#pragma omp parallel private(joe), private(alf), shared(bert), \
    private(i,j,k), shared(fred,n)
```

There are some apparently odd restrictions; some have good reasons, some others do not. An example is that `DO`, `for` and `sections` without `parallel` are **not** allowed to have `shared`. There are more restrictions on `private`, however, but there is no problem with simple code, as in the examples.

- They are very important for practical use, and are described later, under *Critical Guidelines*.

3.1.3 Firstprivate

`firstprivate` is `private` with initialisation; the private objects start with the shared values (i.e. the value in the variable outside the parallel section). Variables are copied as if by assignment, which is particularly important for C++ programmers. There are other forms of `private`, for advanced use only – they are not often useful, and this course does not cover them.

Fortran Example:

```
module P
    integer :: joe = 123, alf = 456
end module P

print *, joe, alf    ! 123 456
!$omp parallel private(joe), firstprivate alf)
print *, joe    ! Undefined value
print *, alf    ! 456
joe = omp_get_thread_num ( )
alf = joe
print *, joe, alf    ! Thread num., twice
!$omp end parallel
print *, joe, alf    ! Both undefined values
```

C/C++ Example:

```
int joe = 123, alf = 456;

printf("%d %d\n",joe,alf);    /* 123 456 */
#pragma omp parallel private(joe), firstprivate(alf)
{
    printf("%d\n",joe);    /* Undefined value */
    printf("%d\n",alf);    /* 456 */
    joe = alf = omp_get_thread_num();
    printf("%d %d\n",joe,alf) ;    /* Thread num., twice */
}
printf("%d\n",joe,alf);    /* Both undefined values */
```

3.1.4 Reductions

These are exactly the same concept as reductions in MPI, and are one of the critical parallel primitives; you can think of them as a summation across threads. They perform some operation over all threads in an unspecified order, using hidden accumulators, and return the aggregate result in the named variable. They are the most common form of shared update access in a well-written program, and you should use them, to avoid a lot of other problems.

OpenMP initialises the variable automatically – this is a ' gotcha', because it is not like serial mode, which obviously does not. You are **strongly** recommended to initialise them yourself, because being able to run in serial mode is important, and you must initialise to OpenMP's value (no other will do), or you will change the meaning of the program between serial and parallel modes.

3.1.5 Fortran Reductions

```
INTEGER FUNCTION Mysum (array)
    INTEGER, INTENT(IN) :: array(:)
    INTEGER :: k, n
    n = 0
!$OMP PARALLEL DO REDUCTION(+:n)
    DO k = 1, UBOUND(array,1)
    n = n + array(k)
    END DO
!$OMP END PARALLEL DO
    Mysum = n
END FUNCTION Mysum
```

This is functionally equivalent to `SUM(array)`, so you would not write it for serial code. Summation is not the only reduction operator, and valid Fortran ones are:

Operator	Initial value
+	0
*	1
-	0
.AND.	.true.
.OR.	.false.
.EQV.	.true.
.NEQV.	.false.
MAX	-HUGE()
MIN	HUGE()
IAND	NOT(0)
IOR	0
IEOR	0

Examples:

```
x = x * (y+1.23)
k = k .OR. (b > 456.789)
z = MAX(z,p-3,q(5))
```

If the clause is something like:

`!$omp parallel do reduction(`*op*`:`*list*`)`

Then the allowed accumulation statements are of the form:

var `=` *var op expression*

Where *op* is the same as in the directive and *var* is in *list*. There are two key points to note:

- *var* must not be used in *expression*.
- In the loop where it is a reduction variable, you must use *var* only for accumulation, and not access it otherwise.

If the clause is something like:

`!$omp parallel do reduction(`*intrinsic*`:`*list*`)`

Then the allowed accumulation statements are of the form:

var `=` *intrinsic*`(`*var*`,`*expression*`,...)`

Where `intrinsic` is the same as in the directive and *var* is in *list*, with the same restrictions on the use of *var*.

3.1.6 C/C++ Reductions

```
int function Mysum (const int * array, int size) {
    int k, n;
    n = 0;
#pragma omp parallel for reduction(+:n)
    for (k = 0; k < size; ++k)
        n += array[k];
    return n;
}
```

Summation is not the only reduction operator, and valid C/C++ ones are:

Operator	Initial value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Note that there is no *max* or *min* in OpenMP 2.5, which is a real pain, though they were added in OpenMP 3.1, and your compiler probably has them, though their syntax was not specified until OpenMP 4.5! Examples are:

```
x *= (y+1.23) ;
k ||= (b > 456.789) ;
z &= (p-3 | q[5]) ;
```

or:

```
x = x * (y+1.23) ;
k = k || (b > 456.789) ;
z = z & (p-3 | q[5]) ;
```

though **probably not**, even in C++:

```
z = max(z , p-3) ;
```

The allowed syntax is given below.

For all operators except *max* and *min*, if the clause is something like:

`#pragma omp parallel for reduction(`*op*`:`*list*`)`

Then the allowed accumulation statements are of the form:

var `=` *var op expression*

var op`=` *expression*

var`++, ++`*var*`,` *var*`--, --`*var*

Where *op* is the same as in the directive and *var* is in *list*. Obviously, both forms of `++` in the accumulation must match `+` in the directive, and similarly for `--` and `-`.

max and *min* are a bit of a problem. The experts' and compilers' consensus for OpenMP 3.1 was to use the following syntax, even for C++:

```
    #pragma omp parallel for reduction(min:list)
          if ( var > expression ) var = expression ;
#pragma omp parallel for reduction(max:list)
          if ( var < expression ) var = expression ;
```

However, OpenMP 4.5 specifies:

```
    #pragma omp parallel for reduction(min:list)
          var = var > expression ?  expression :  var ;
          var = var > expression ?  expression :  var ;
```

Compilers seem to allow both, and may well allow other forms.

There are three key points to note:

- *var* must not be used in *expression*.
- In the loop where it is a reduction variable, you must use *var* only for accumulation, and not access it otherwise.

- You must **not** use the result of the accumulation as an expression; e.g. `x = ++y` is not allowed if `y` is a reduction variable.

3.1.7 Debugging and Tuning

Most of this course is how to avoid the need for debugging, because that is such an evil task for OpenMP, but one aspect is so critical that it needs mentioning now. Explaining the reasons is left until later.

- Erronous code usually **appears** to work.

Most failures occur only rarely, in large problems or in only some implementations, so it is very important not to assume that bugs will always show up. This is why I regard shared-memory parallel debugging as so much harder than serial debugging, and even harder than message-passing like MPI, even though it looks easier.

Almost all tuning information is left until later but, again, one aspect is so critical that it needs mentioning now. It also applies to the tuning of serial programs, but it is redoubled in spades for shared-memory parallelism. It can mean a factor of a hundred slowdown but, more commonly, expect a factor of up to about ten.

- The key to shared memory performance is caching.

On almost all current systems, memory is divided into cache line units, typically between 32 and 128 bytes long, possibly even 256, aligned according to their size. The CPU loads and stores whole cache lines only, even if it is using only one byte in a line. All CPUs can read the same cache line at the same time, but precisely one must own it in order to write to it. It the writing core does not own it, the cache line must be moved to that core.

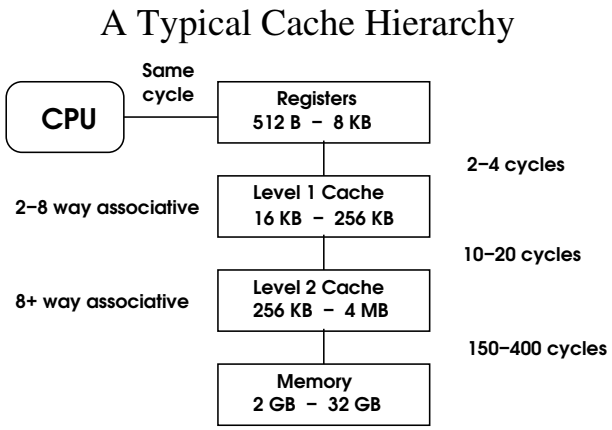

Figure 1.1

The hardware usually has direct cache-to-cache links, but transferring data over them still takes time and it is very easy to overload them, which leads to cache thrashing and dire performance.

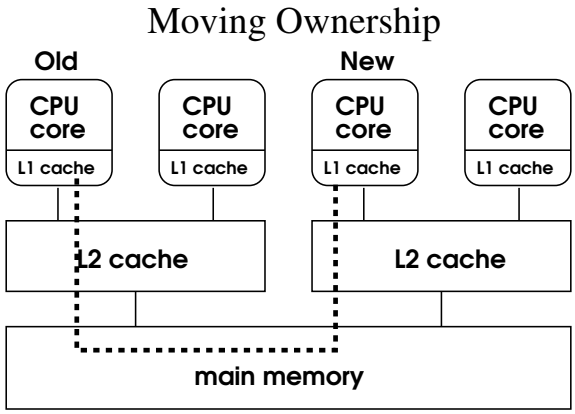

Figure 1.2

- Each thread's data should be well separated.

Remember that cache lines are 32–256 bytes long, and do not bother for occasional accesses. Accessing locations in the same line in parallel works – it just runs very slowly. Even a factor of 100 slowdown 0.01% of the time does not matter.

As an example of the problem, calculate $\tilde{V} = a\,.\,\tilde{V} + c$ for a vector $\tilde{V}$, using separate threads to copy the even and odd elements.

C/C++ example:

Thread 1:
```
for (k = 0; k < n; k += 2) V[k] = a*V[k]+c;
```
Thread 2:
```
for (k = 1; k < n; k += 2) V[k] = a*V[k]+c;
```

Fortran example:

Thread 1:
```
DO k = 1,n,2
    V(k) = a*V(k)+c
END DO
```
Thread 2:
```
DO k = 2,n,2
    V(k) = a*V(k)+c
END DO
```

Another example of how this affects OpenMP is a matrix copy – this one is bad, and we need to reverse the order of either the loops or indices – it does not matter which.

Fortran example:

```
REAL(KIND=DP) :: here(:,:), there(:,:)
!$OMP PARALLEL DO
    DO m = 1, UBOUND(here,1)
        DO n = 1, UBOUND(here,2)
            there(m,n) = here(m,n)
        END DO
    END DO
```

```
!$OMP END PARALLEL DO
```

C/C++ example:

```
double here[size1][size2], there[size1][size2];
#pragma omp parallel for
    for (n = 0; n < size_2; ++n)
        for (m = 0; m < size_1; ++m)
           there[m][n] = here[m][n];
#pragma omp end parallel for
```

3.1.8 Conclusion

That is essentially **all** that you need for simple SIMD work, not just for doing the examples, but for writing real programs. Of course, the devil is in the details, and writing efficient SIMD code is rarely as easy as adding the directives.

We have not yet covered what **not** to do, and we shall return to that after covering simple SPMD. Nor have we covered calling procedures in SIMD loops – i.e. Fortran subroutines and Fortran/C/C++ functions. And there are a small number of other useful features, which are needed only as you move on to more advanced SIMD work.