Introduction to OpenMP

More Syntax and SIMD

Nick Maclaren

nmm1@cam.ac.uk

September 2019

C/C++ Parallel for (1)

I said that I would give the precise rules later

for ([<type>] <var> = <expr> ;
<var> <relop> <expr> ;
<increment expression>)

<increment expression> can be:

<var>++, ++<var>, <var>--, --<var>,
<var> += <expr>, <var> = <var>+<expr>,
<var> -= <expr>, <var> = <var>-<expr>

C/C++ Parallel for (2)

The constraints are more like Fortran than C/C++:

- <var> must be a signed integer variable
  3.0 relaxed this, but it's a good rule

- <relop> is one of the relational operators

- Each <expr> must be invariant over the loop
- Don't include any side effects in them

I recommend using only really simple expressions
If in doubt, assign to variables and use those

Scheduling Clause

You can specify the scheduling for each loop
Use it on the DO/for–loop directives
This is OpenMP scheduling policy, not system

For normal SIMD work, use schedule(static)
Specifying it explicitly means compiler knows

This divides the loop into equal chunks
Then hands each chunk to a single thread

Other schedule options are described later

Multiple Loops

You can parallelise multiple consecutive loops
collapse(N) specifies N loops
The order is the same as serial execution

- No other intervening statements are allowed

Probably OK with comments, but avoid them, anyway

- No loop controls depend on an outer loop

C/C++ Example

```
#pragma omp for collapse(3), reduction(*:x)
for (i = 0; i < l; ++i)
    for (j = 0; j < m; ++j)
        for (k = 0; k < n; ++k) {
            x += array[i][j][k];
        }
    }
}
```

Data Environment Clauses (1)

Allowed on most parallel or work−sharing constructs

Most have the syntax <keyword>(<list>)
<list> is a list of variable names

Most (inc. shared and private) can be repeated
Mustn't repeat any variable name, of course

```
#pragma omp parallel default ( none ) , private ( joe ) ,    \
      private ( alf ) , shared ( bert ) ,    \
      private ( i , j , k ) , shared ( fred , n )
```

Data Environment Clauses (2)

There are some apparently odd restrictions
Some have good reasons, some others don't

E.g. DO/for/sections without parallel are
not allowed to have shared

There are more restrictions on private, however
No problem with simple code, as in examples

- But they are very important for practical use

Described later, under critical guidelines

Firstprivate

firstprivate is private with initialisation
The private objects start with the shared values

Variables are copied as if by assignment
Fortran allocatable variables need 3.0

Aside: copy constructor called in which thread(s)?
This sort of ambiguity is why using C++ is a problem

Other forms of private, for advanced use only
Not often useful, and this course doesn't 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    ! Undefined values in 2.5
```

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 ) ;    // Undefined values in 2.5
```

OpenMP 3.0

From 3.0 the shared variable value is preserved

- I strongly advise not relying on that

The shared/private difference is confusing enough
And some potential gotchas I am not describing

Reductions (1)

Exactly the same as reductions in MPI

- One of the critical parallel primitives

Think of a summation across threads

They perform some operation over all threads
In an unspecified order, using hidden accumulators
Return the aggregate result in the named variable

Most common form of shared update access

- Use them, and avoid a lot of other problems

Reductions (2)

OpenMP initialises the variable automatically
A 'gotcha', because it is not like serial mode

- Strongly recommended to initialise yourself

Being able to run in serial mode is important

- Must initialise to OpenMP's value (no other)

Or will change meaning of program between modes

Fortran Example

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

This is equivalent to SUM(array)

Fortran Reductions (1)

Operator	Initial value
+	0
*	1
–	0
.AND.	.true.
.OR.	.false.
.EQV.	.true.
.NEQV.	.false.
MAX	–HUGE()
MIN	HUGE()

Fortran Reductions (2)

Operator	Initial value
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) )
```

Fortran Accumulation Forms (1)

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

Then the allowed accumulation statements are:

```
<var> = <var> <op> <expression>
```

Where <op> is the same and <var> is in <list>

- <var> must not be used in <expression>
- Use <var> only for accumulation

Fortran Accumulation Forms (2)

!$omp parallel do reduction(<intrinsic>:<list>)

Then the allowed accumulation statements are:

<var> = <intrinsic>(<var>,<expression>,...)

Where <intrinsic> is the same and <var> is in <list>

With the same restrictions on the use of <var>

C/C++ Example

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

C/C++ Reductions (1)

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

C/C++ Reductions (2)

Operator	Initial value
max	–infinity
min	+infinity

And the equivalent extreme value for integers

Note no max or min in 2.5 – a real pain
Came in 3.1, so compilers probably have them
⇒ But no specification of syntax until 4.5!

C/C++ Reductions (3)

Examples:

```
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] ) ;
```

Probably not, even in C++: `z = max ( z , p-3 ) ;`
See later what syntax IS allowed

C/C++ Accumulation Forms (1)

#pragma omp parallel for reduction(<op>:<list>)

Then the allowed accumulation statements are:

<var> <op>= <expression>
<var> = <var> <op> <expression>
<var>++, ++<var>, <var>−−, −−<var>

Where <op> is the same and <var> is in <list>

- <var> must not be used in <expression>
- Use the variable only for accumulation
- Don’t use the result of the accumulation

C/C++ Accumulation Forms (2)

#pragma parallel for reduction(<min,max>:<list>)

The experts' and compilers' consensus for 3.1:

```
if ( <var> > <expr> ) <var> = <expr> ;
if ( <var> < <expr> ) <var> = <expr> ;
```

According to 4.5:

```
<var> = <var> > <expr> ? <expr> : <var> ;
<var> = <var> < <expr> ? <expr> : <var> ;
```

With the same restrictions on the use of <var>

%deity alone knows what else compilers accept

Debugging

- Most of this is how to avoid the need for debugging

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
Don't assume that bugs will always show up

- It is why I regard SMP debugging as hard

It only looks easier than, say, MPI

Tuning

- Almost all tuning information is left until later

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 SMP work

- It can mean a factor of 100 slowdown

More commonly, expect a factor of up to 10 or so

Must Think Caching

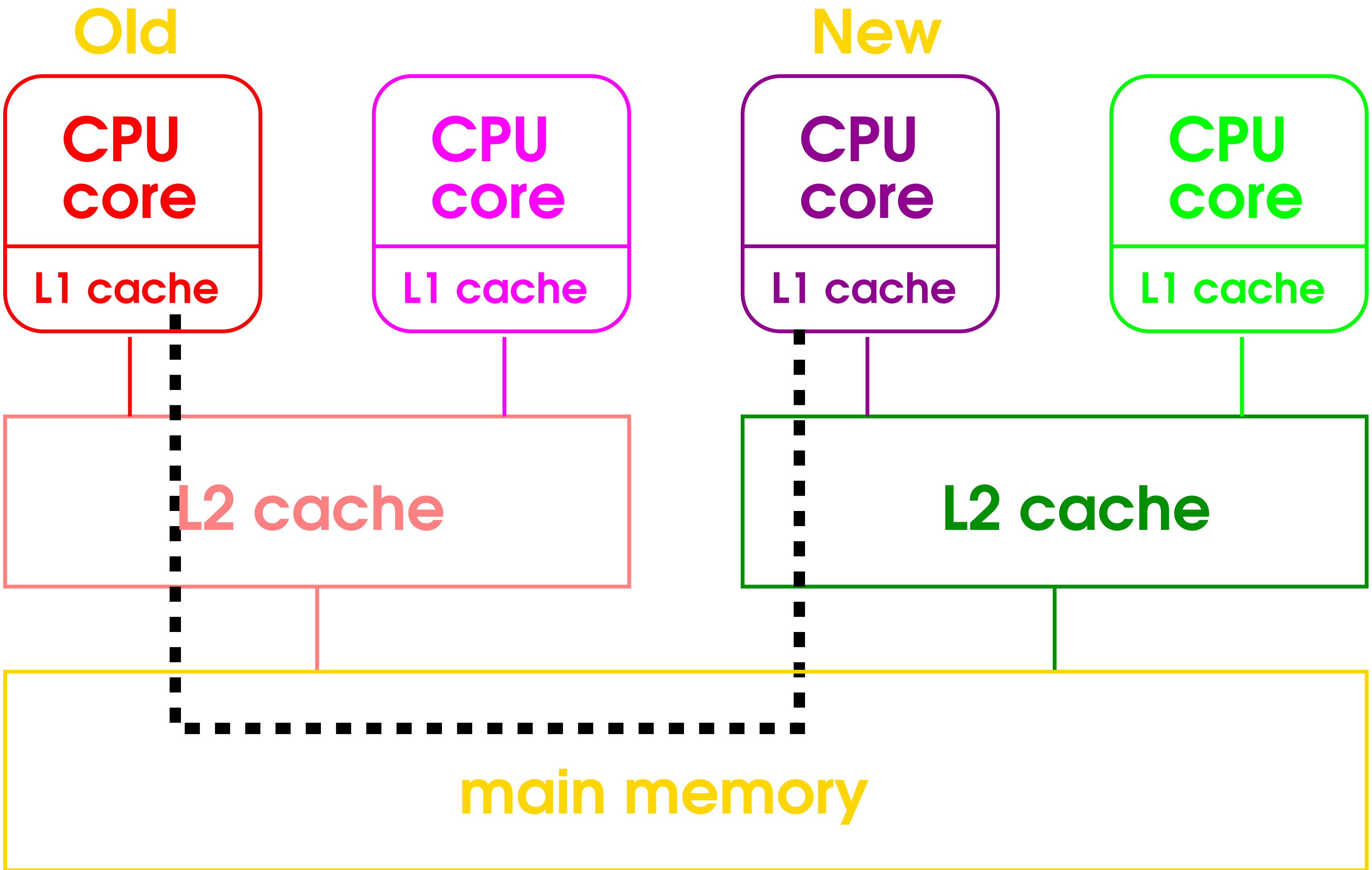
The key to shared memory performance is caching

- All memory is divided into cache line units
Typically 32–128 bytes, aligned according to its size

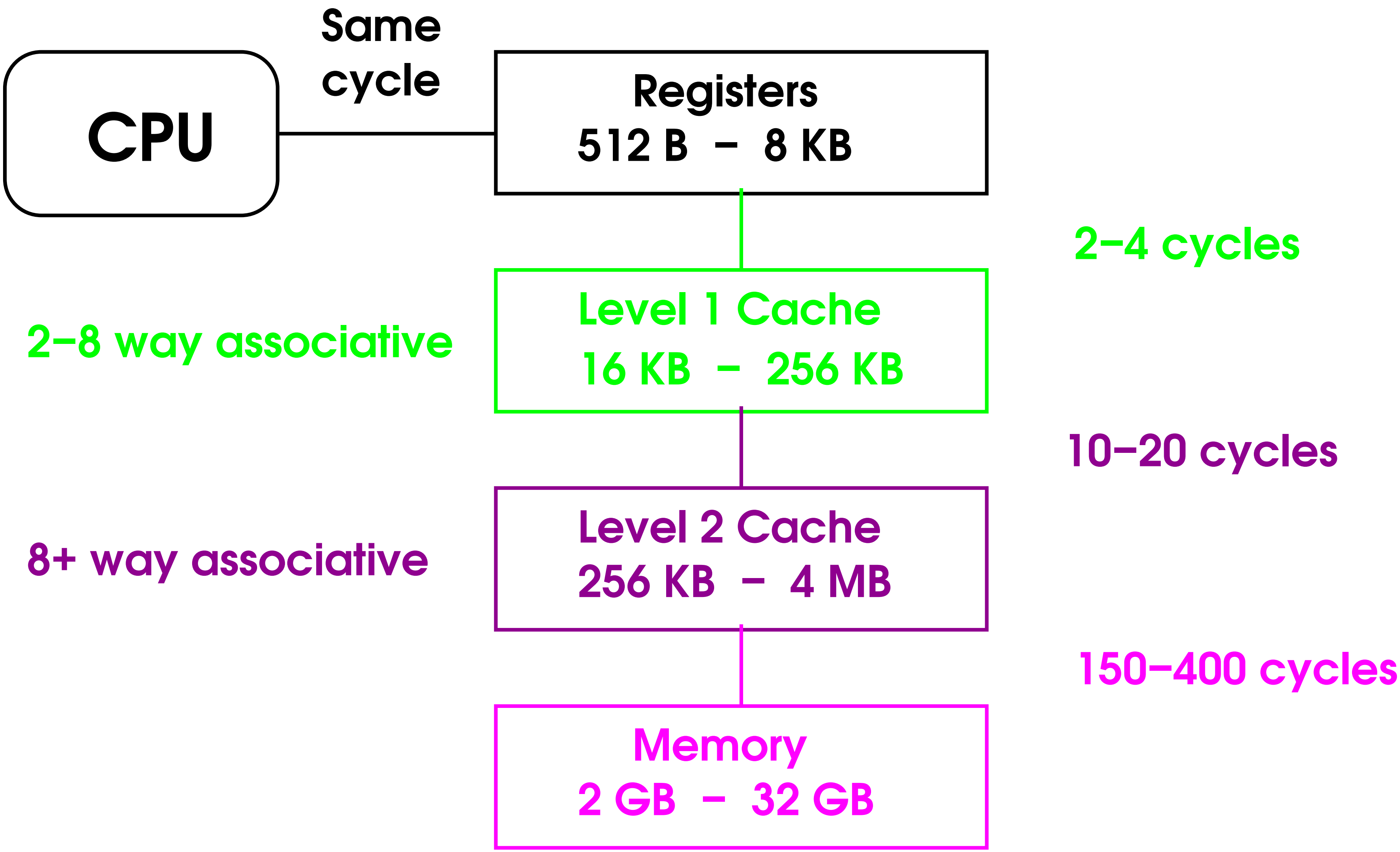
- The CPU loads and stores whole cache lines only
Even if it is using only one byte in a line

All CPUs can read the same cache line
- Precisely one must own it to write to it
It it doesn’t, the cache line must be moved to it

Moving Ownership



A Typical Cache Hierarchy



Cache Line Sharing

The hardware usually has direct cache–cache links

- But they take time, and it's easy to overload them

Leads to cache thrashing and dire performance

- Each thread's data should be well separated

xRemember cache lines are 32–256 bytes long

- Don't bother for occasional accesses

The code works – it just runs very slowly

100$\times$ slowdown 0.01% of the time doesn't matter

Example of Problem

Calculate $\tilde{V} = a\,.\tilde{V} + c$ for a vector $\tilde{V}$
Using separate threads for even and odd elements

Thread 1: [C/C++: for (k = 0 ; k < n ; k += 2)]

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

Thread 2: [C/C++: for (k = 1 ; k < n ; k += 2)]

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

Fortran Example

Consider a matrix copy – this one is bad
Need to reverse the order of the loops (or indices)

```
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

Consider a matrix copy – this one is bad
Need to reverse the order of the loops (or indices)

```
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
```

That's It, Really

- That all you need for simple SIMD work

Not just for the examples, but for real programs

- We haven't yet covered what NOT to do

We shall return to that after covering simple SPMD

- Nor 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
Needed only as you do more advanced SIMD work