Introduction to OpenMP

Simple SPMD etc.

Nick Maclaren

nmm1@cam.ac.uk

September 2019

Terminology

I am badly abusing the term SPMD – tough
The original meaning makes little sense nowadays

In this course, I mean a type of program design

SPMD Includes SIMD

SPMD proper is a superset of SIMD
Going to cover some of the non–SIMD aspect

- But there isn't a rigid boundary between the two

Some SPMD features are useful for SIMD
Scheduling for irregular loops is one example

OpenMP library functions are another example
They are useful for producing good diagnostics

The Great Myth

Many books and Web pages say SPMD is simple

They could not be more wrong

- It is possible to use SPMD very simply
- But easy to write dangerous code by mistake

Applies to both correctness and performance

- This course describes some simple, safe rules

Simplest SPMD Model

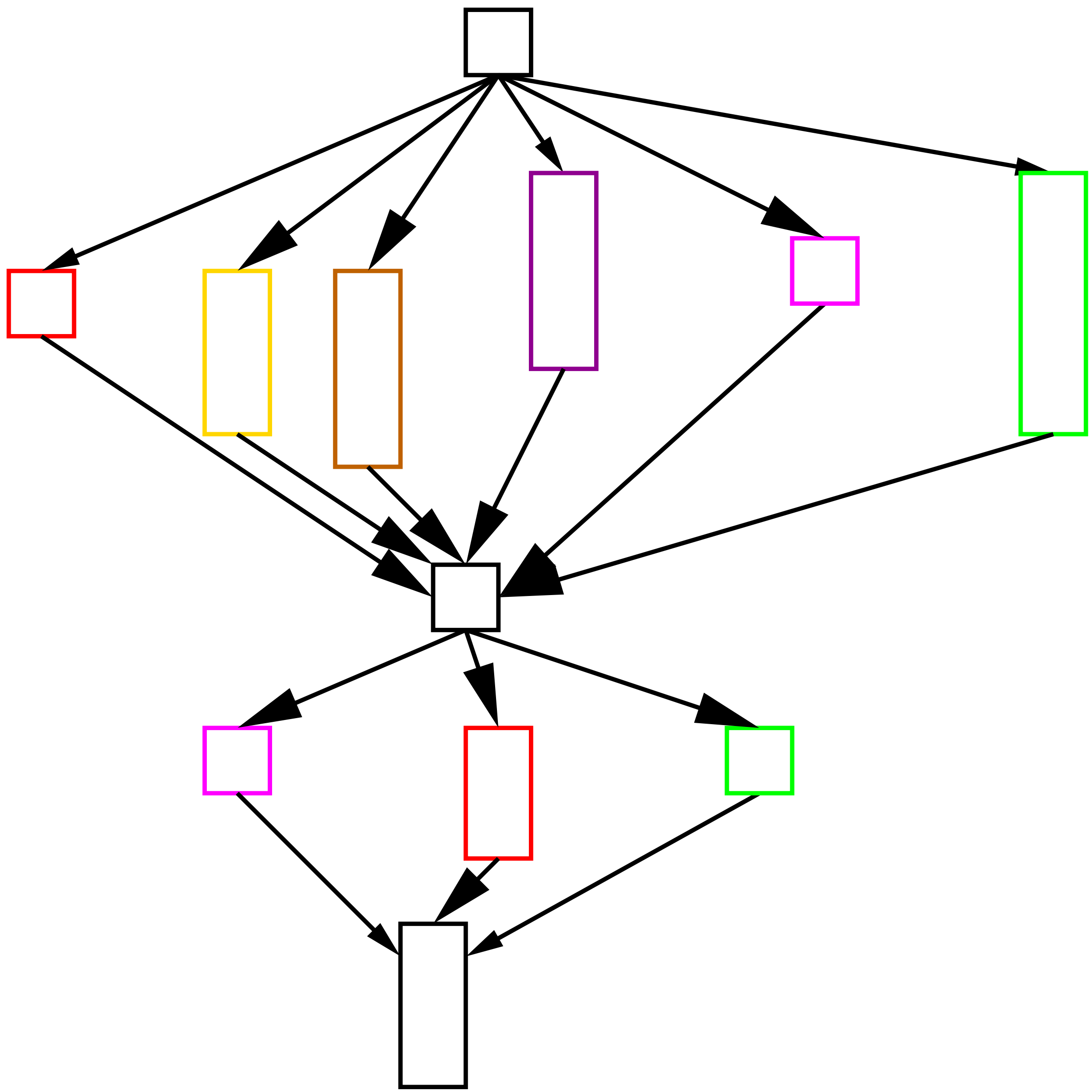
At any point, one thread starts a parallel region
Each subthread runs to completion, and finishes
The serial code then carries on executing

In the simple model we are considering:

- No communication between threads
- All global data is read–only
  Reductions are allowed, of course

Beyond that, there be dragons ...

Simple SPMD Task Structure



Just Do It

Don't need anything more for SPMD programming
All you need is a plain parallel directive
Then just select on thread number in your code

- Can be tricky to adapt to different core counts

Not clever to code for precisely N threads

Trivial example is coding your own parallel DO/for
Futile, unless you cannot use DO/for
For example, if data is held in a linked list

Just Do It (Fortran)

```
REAL ( KIND = KIND ( 0.0D0 ) ) :: array ( size )
INTEGER :: chunk , index

! This rounds the chunk size up
chunk = ( size - 1 ) / omp_num_threads ( ) + 1
!$OMP PARALLEL private ( index )
DO index = chunk * omp_thread_num ( ) + 1 ,    &
      MIN ( chunk * ( omp_thread_num ( ) + 1 ) , size )
   . . .
END DO
!$OMP END PARALLEL
```

Just Do It (C/C++)

```
double * array ;    /* size elements */
int chunk , index ;

/* This rounds the chunk size up */
chunk = ( size - 1 ) / omp_num_threads ( ) + 1 ;
#pragma omp parallel private ( index )
for ( index = chunk * omp_thread_num ( ) ;
        index < chunk * ( omp_thread_num ( ) + 1 ) &&
                index < size ;
        ++ index ) {
    . . .
}
```

Basic SPMD Directive

Adding directives for SPMD is very simple

The basic one is sections, for parallel tasks
It's a bit like a parallel SELECT CASE or switch

There are both work–sharing and combined forms
We shall use the combined form in examples

- It's not very useful – OpenMP tasks are later

The Just Do It method is more flexible

Omitted here – see the notes

Fortran Example

```
!$OMP PARALLEL SECTIONS [ clauses ]
    !$OMP SECTION
        < code of structured block >
    !$OMP SECTION
        < code of structured block >
    !$OMP SECTION
        < code of structured block >
!$OMP END PARALLEL SECTIONS [ clauses ]
```

Each section is potentially executed in parallel

C/C++ Example

```
#pragma omp parallel sections [ clauses ]
        #pragma omp section
                < code of structured block >
        #pragma omp section
                < code of structured block >
        #pragma omp section
                < code of structured block >
#pragma omp end parallel sections [ clauses ]
```

Each section is potentially executed in parallel

Starting SPMD (1)

Not much more to say about the sections directive
Its clauses are the usual data environment ones

- Each section will run in a separate thread

Scheduling sections to threads is unspecified

It is permitted to have more sections than threads
Unspecified behaviour makes it very hard to tune

- Generally, use only as many sections as threads

And only as many threads as cores (or fewer)

Starting SPMD (2)

Not recommended for a dynamic number of threads

- Can equally well use parallel DO/for to start

Same worker function with different arguments
Or each iteration can call a separate procedure

Isn't a rigid boundary between SIMD and SPMD

- Difference is in your approach to the problem

Skilled programmers should have no problem
Feel free to use loops if you are happy to do so

Library Functions (1)

The ones that obtain information are perfectly safe
You can use them almost anywhere, without problems

```
double omp_get_wtime ( void ) ;
REAL (KIND = KIND ( 0.0D0 ) )    &
        FUNCTION omp_get_wtime ( )
```

The wall–clock time in seconds

omp_get_wtick – precision of time

Exactly the same syntax as omp_get_wtime
You probably won't find it useful, but it's there

Library Functions (2)

```
int omp_get_num_threads ( void ) ;
INTEGER FUNCTION OMP_GET_NUM_THREADS ( )
```

The number of threads in the current team

```
int omp_get_thread_num ( void ) ;
INTEGER FUNCTION OMP_GET_THREAD_NUM ( )
```

The index of the current thread

```
int omp_in_parallel ( void ) ;
LOGICAL FUNCTION OMP_IN_PARALLEL ( )
```

True if in a parallel region, false otherwise

Usual language meanings of true and false

Environment Variables

For SPMD, different ones are better:

```
export OMP_SCHEDULE=dynamic
export OMP_DYNAMIC=true
```

Can also use schedule(dynamic) clause

- But, as always, they may not always be best

There's too much that's implementation−dependent

- OMP_NUM_THREADS used exactly as for SIMD

Threadprivate (1)

A global or static variable private to a thread

- Each thread has a separate copy

- Put it immediately after the variable's declaration

It must be in the same scope as the declaration

Must occur before any references to the variable

Must have a global lifetime (static or SAVE)

- Obviously, don't specify it for arguments!

Or other inherited variables (in any language)

Threadprivate (2)

- The master thread 0's copy is permanent
It's also accessible from serial code, with care

- Otherwise access only from its owning thread
Use all of these only within a parallel region
They may become undefined on entry and exit
Ensuring that they don't is seriously advanced use

- Don't put variables in data environment clauses

Threadprivate (3)

Most other restrictions forbid unimplementable uses
You will probably never have trouble

- Provided you do only what this course teaches

And you don't use it together with tasks

E.g. the parallel region mustn't call a SMP library
Threadprivate isn't safe with nested parallelism
This minefield is described later

Threadprivate (Fortran)

```
REAL ( KIND = dp ) , SAVE , ALLOCATABLE :: array ( : , : )
 REAL ( KIND = dp ) , SAVE :: vector ( 5 ) , var
 !$OMP THREADPRIVATE ( array , vector , var )

 < Allocate and use array, vector and var >
```

- Note no !$OMP END THREADPRIVATE

Any reasonable type and declaration is allowed

Should be in modules, initialised or use SAVE

- Don’t use with COMMON or EQUIVALENCE

Threadprivate (C/C++)

```
static double array [ 5 ] [ 5 ] , * ptr ;
static int index = 123 ;
#pragma omp threadprivate ( array , index , ptr )

< Use array, index and ptr >
```

Any reasonable type and declaration is allowed

Should be file– or namespace–scope or static

- extern must always use it or never use it
- Must be copyable if declared with an initialiser

Copyin

The copyin clause is very like firstprivate
Copies from the master thread zero to all threads

- It can be used only on threadprivate variables
- It can be used only on parallel directives

Fortran allocatable variables need 3.0

Not very useful in OpenMP subset taught here
No examples given, as used exactly like firstprivate

Performance

Keep it simple and you will rarely have problems

- Try to avoid having to tune SPMD code

- Keep each thread's data as separate as possible

Remember the caching? That's the critical aspect

- Make each parallel section fairly long

That's in terms of execution time, not lines of code

- Try to share work equally between threads

Easy for schedule(dynamic) with high loop count

Tuning

This can be from simple to diabolical
It depends on how threads are scheduled

- And that's unspecified and unpredictable

If some code is half memory– and half CPU–limited
Performance will be bad if all of one type runs at once
And good if there is a mixture at all times

Similarly with accessing different regions of data
Very common with some classes of application

Problem Loops

Some loops access data in cache−hostile ways
And aren't practical to rearrange or reorder

SIMD assumes iterations are homogeneous
Each one takes roughly the same time to complete

- Sometimes that isn't even remotely true

Some OpenMP facilities that can help with these
Generally, avoid them unless you really need them

- Remember that each loop can be different

System/Kernel Scheduling

System scheduling chooses which threads to run
Often called kernel or thread scheduling

- Not controllable by the ordinary programmer

Most POSIX facilities to do it don't actually work
Administrator does it through system configuration

- Close your eyes and hope that it works

If not, must work together with system administrator
Most techniques taught in this course are robust

Scheduling (1)

OpenMP scheduling distributes work between threads

- Can be used on DO/for construct only

Only form of scheduling taught in this course

schedule(static) == best default for SIMD
loops are divided into equal chunks

- Essential if logic depends on communication

This course doesn't cover such advanced use

Scheduling (2)

schedule(static, size) divides into size chunks
Assigned to threads in a round robin fashion

May help with some cache conflict problems

- Try using schedule(static, 1) and see if it helps

Can also be used to expose some race conditions
If it fails, there is a bug in the data usage
I used it for that when testing my specimen answers

Scheduling (3)

The following pairs of loops behave similarly:

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
DO i = omp_get_thread_num()+1, limit,    &
       omp_get_num_threads()

!$OMP PARALLEL DO SCHEDULE(STATIC,1)
DO i = 1, limit

#pragma omp for schedule(static)
for ( i = omp_get_thread_num() ; i < limit ;
       i += omp_get_num_threads() ) ...

#pragma omp for schedule(static,1)
for ( i = 0 ; i < limit ; ++i ) ...
```

Scheduling (4)

schedule(dynamic)
Each thread takes a single iteration of the loop
As each thread finishes, it takes another iteration
Consider when iterations vary a lot in time taken

schedule(dynamic, size)
Threads take chunks of size iterations

- This might help for non–uniform loops

Don’t make the chunk size too small, though

Scheduling (5)

This is mentioned mainly for completeness
I advise trying it only as a last resort

schedule(guided [, size])
An adaptive algorithm, a bit too complex to describe

Size is the minimum chunk size (default 1)

If you try it, start with size omitted

Thread Synchronisation

Mainly ensuring some code is executed serially

- That is a restricted form of synchronisation

It is also needed for SIMD (e.g. for I/O)

Those facilities are covered in the next lecture

Thread communication is often essential

That includes between the master and a worker

- This is not really covered in this course

Some facilities are mentioned, but no more