Introduction to OpenMP

Simple SPMD etc.

N.M. Maclaren

Computing Service

nmm1@cam.ac.uk
ext. 34761

August 2011

1.1 Minimal SPMD

SPMD proper is a superset of SIMD, and we are now going to cover some of the non-SIMD aspect, but there is not a rigid boundary between the two, some SPMD features are useful for SIMD. Scheduling for irregular loops is one example, and OpenMP library functions are another example, because they are useful for producing good diagnostics.

Many books and Web pages say *SPMD* is simple.

- **They could not be more wrong**.

While it is possible to use SPMD very simply, it is easy to write dangerous code by mistake, and this applies to both correctness and performance. This course describes some simple, safe rules, and does not attempt to describe SPMD in general.

The simplest SPMD model is that, at any point, one thread starts a parallel region. Each subthread runs to completion, and then finishes. The serial code then carries on executing. In the simple model we are considering, there is:

- No communication between threads.
- All global data is read-only; reductions are allowed, of course.

Beyond that, there be dragons . . . .

Simple SPMD Task Structure

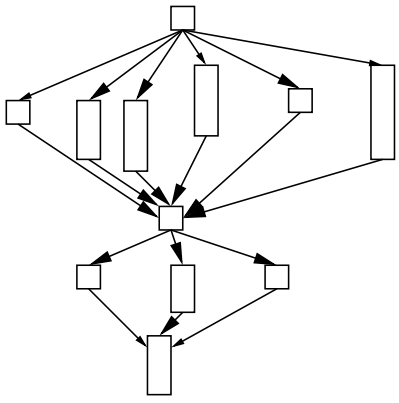

Figure 1.1

You do not actually need anything more than we have covered for SPMD programming, because all you need is a plain `parallel` directive, and then you can just select on the thread number in your code. This can be tricky to adapt to different core counts, and it is not clever to code for a fixed number of threads, as you will want to use more in the future. That is why this course does not teach this technique first, but starts with SIMD.

The most trivial example is coding your own parallel `DO` or `for` loop. This is obviously futile, **unless** you cannot use OpenMP's parallelisation of the loops because of a fundamental data dependency between iterations. For example, if your data is held in a linked list, then moving to the next node is such a dependency; this is the simplest way of parallelising such loops.

Fortran Example:

```
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
```

C/C++ Example:

```
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) {
    . . .
}
```

1.2 Basic SPMD Directive

Adding the directives for SPMD is very simple . The basic one is `sections`, for parallel tasks; it is a bit like a parallel `SELECT CASE` or `switch`. There are both work-sharing and combined forms, and we shall use the combined form in examples.

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 ]
```

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 ]
```

In both cases, each section is potentially executed in parallel. There is really not much more to say about the `sections` directive; its clauses are just the usual data environment ones, as on `DO/for` except that `schedule` is not allowed.

Each section will run in a separate thread, and the scheduling of sections to threads is unspecified. While it is permitted to have more sections than threads, the unspecified behaviour makes it very hard to tune. Generally, you should use only as many sections as threads, and only as many threads as cores (or fewer).

This is not very useful for a dynamic number of threads, but you can equally well use a parallel `DO/for` to start different threads, and call the same worker function with different arguments in each iteration, or each iteration can call a separate procedure. There is no rigid boundary between SIMD and SPMD, and the difference is in your approach to the problem. Skilled programmers should have no problem, so feel free to use loops for the examples if you are happy to do so.

1.3 Library Functions

The ones that obtain information are perfectly safe, and you can use them almost anywhere, without problems. We have covered some of them already, but will go through them.

C/C++: `double omp_get_wtime (void);`
Fortran: `REAL(KIND=KIND(0.0D0)) FUNCTION OMP_GET_WTIME ()`

This returns the wall-clock time (i.e. real time, physical time or whatever you like to call it) in seconds, from an unspecified starting point.

`omp_get_wtick` returns the precision of `omp_get_wtime`, and has exactly the same syntax as `omp_get_wtime`. You probably will not find it useful, but it is there.

C/C++: `int omp_get_num_threads (void);`
Fortran: `INTEGER FUNCTION OMP_GET_NUM_THREADS ()`

This returns the number of threads in the current team which, in the simple way we are using OpenMP, should be the number you started it with. It is useful for writing code that adapts to the number of threads.

C/C++: `int omp_get_thread_num (void);`
Fortran: `INTEGER FUNCTION OMP_GET_THREAD_NUM ()`

This returns the index of the current thread, in the range zero to one less than the number of threads. It is useful for for diagnostics, and for more advanced SPMD.

C/C++: `int omp_in_parallel (void);`
Fortran: `LOGICAL FUNCTION OMP_IN_PARALLEL ()`

This returns true if in a parallel region, and false otherwise, with the usual language meanings of true and false. It is useful for internal error checking (e.g. in procedures that should be called only in serial mode) and diagnostics.

1.4 Environment Variables

For SPMD, different ones from the SIMD recommendations are better:

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

You can also use the `schedule(dynamic)` clause on `DO/for` loops. But, as always, they may not always be the best ones to use. There is just too much that is implementation-dependent, and even dependent on the details of your particular system. `OMP_NUM_THREADS` is used exactly as for SIMD.

1.5 Threadprivate

This declares a global or static variable to be private to a thread, and each thread has a separate copy. You should put it immediately after the variable's declaration. It must be in the same scope as the declaration, and it must occur before any references to the variable, but it also has the following constraints:

- It must have a global lifetime; i.e. it must be a global declaration at file scope or in a module, or a local declaration with `static` or `SAVE`.
- Obviously, do not specify it for arguments, or other inherited variables (in any language). It must be specified together with the declaration that creates the variable.

Unlike ordinary `private`, the master thread's copy is permanent, and is also accessible from serial code, but there are a significant number of *gotchas* (mentioned later). Otherwise you must access it only from its owning thread. It is best to use both ordinary

`private` and `threadprivate` variables only within a parallel region, because they may become undefined on entry and exit from one; ensuring that they do not is seriously advanced use.

- You must not put `threadprivate` variables in data environment clauses.

Most of the other restrictions forbid unimplementable uses, and you will probably never have trouble if you do not rely on the master thread's value being accessible in serial mode.

Fortran example:

```
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 that there is no `!$OMP END THREADPRIVATE`. Any reasonable type and declaration is allowed and, as mentioned, that should be in modules, use initialisation (which implies `SAVE`) or use `SAVE`. Do not use it with `COMMON`, variables in `COMMON`, or variables in `EQUIVALENCE`.

C/C++ example:

```
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 and,as mentioned, that should be file- or namespace-scope or use `static`. `extern` must always use it or never use it, but `extern` is a *gotcha* it itself, as its specification is truly ming-boggling and is not what most books and Web pages say that it is. There is a specific C++ constraint:

- The class must be copyable if it was declared with an initialiser, for obvious reasons if you think about it!

The `copyin` clause is very like the `firstprivate` clause, and copies from the master thread (zero) to all of the threads. Note that it can be used **only** on threadprivate variables (and not on ordinary private ones), and it can be used only on `parallel` directives or a combined directive. Also, it cannot be used for Fortran allocatable variables, for some reason. It is not very useful in subset of OpenMP covered in this course, and no examples of its use are given, as it is used exactly like `firstprivate`.

1.6 Performance

Keep it simple and you will rarely have problems, and try to avoid having to tune SPMD code. The following are the main guidelines:

- Keep each thread's data as separate as possible. Remember the caching description? This is the most critical aspect.
- Make each parallel section fairly long, in terms of execution time, not lines of code. This is to minimise the overheads of thread switching.

- Try to share work equally between threads, which is easiest if you use dynamic scheduling with a high loop count.

Tuning can be from simple to diabolical, and depends very much on how threads are scheduled, which is unspecified and unpredictable. If some parts of the program are memory-limited and some parts CPU-limited, the performance will be bad if all of one type runs at once, and good if there is a mixture at all times. The same applies when accessing different regions of data, which is very common with some classes of application.

Some `DO/for` loops access data in cache-hostile ways, and are impractical to rearrange or reorder. The SIMD approach assumes iterations are homogeneous, and each one takes roughly the same time to complete, but sometimes that is not even remotely true. Some of the OpenMP SPMD facilities can help with these loops but, generally, avoid them unless you really need them.

- Remember that each loop can be different.

1.7 Scheduling

System scheduling chooses which threads to run, and it is often called kernel or thread scheduling. It is not controllable by the ordinary programmer, and most of the POSIX facilities to do it do not actually work, in practice. In most systems, the administrator does it through the system configuration.

You are recommended to close your eyes and hope that it works; it it does not, you must work together with the system administrator to improve the situation. Most of the techniques taught in this course are robust against the normal variants of system scheduling that you will encounter.

OpenMP scheduling distributes work between threads, can be used on the `DO/for` construct only, and is the only form of scheduling taught in this course.

As mentioned, `schedule(static)`, where loops are divided into equal chunks, is the best default for SIMD. It is also essential if the logic depends on explicit thread communication, but this course does not cover such advanced use.

`schedule(static,size)` divides the loop is `size` chunks, which are assigned to threads in a round robin fashion. Using this may help with some cache conflict problems, or may make them much worse. If you suspect those, try using `schedule(static,1)` and see if it helps or makes things worse. It can also be used to expose some race conditions because, if it fails, there is a bug in the data usage. I used it for that when testing and debugging the examples.

With `schedule(dynamic)`, each thread takes a single iteration of the loop and, as each thread finishes, it takes another iteration. Consider using this when iterations vary a lot in the time that they take. It can also be used when using more threads than cores, but that is advanced use.

That can have very high overheads, when you should use `schedule(dynamic,size)` instead; like the static form, threads take chunks of `size` iterations instead of a single iteration. This is probably the best option for non-uniform loops, but do not make the chunk size too small, so experiment with varying the size.

The last option is mentioned mainly for completeness, and I advise trying it only as a last resort. `schedule(guided[,size])` is an adaptive algorithm, a bit too complex to describe here. `size` is the minimum chunk siz, and the default is one. If you try it, start with `size` omitted.

1.8 Thread Synchronisation

The objective is mainly to ensure that some code is executed serially, which is a restricted form of synchronisation, and is also needed for SIMD (e.g. for I/O). Those facilities are covered in the next lecture.

Thread communication is often essential, and that includes between the master and a worker in a master/worker design. That is **not** really covered in this course – some facilities are mentioned, but no more.