OPENMP TIPS, TRICKS AND GOTCHAS

Mark Bull
EPCC, University of Edinburgh (and OpenMP ARB)
`markb@epcc.ed.ac.uk`

OpenMPCon 2015

A bit of background

- I’ve been teaching OpenMP for over 15 years to students and scientists
- I was chair of the OpenMP Language Committee for about 5 years
- I answer questions on the OpenMP Forum (**`openmp.org/forum`**)
- This is a collection of some accumulated wisdom in particular which you won’t find in most tutorials......
- I hope you might find some of it helpful!

Directives

- Mistyping the sentinel (e.g. `!OMP` or `#pragma opm`) typically raises no error message.
  - Be careful!
  - Extra nasty if it is e.g. `#pragma opm atomic` – race condition!
  - Write a script to search your code for your common typos

Writing code that works without OpenMP too

- The macro `_OPENMP` is defined if code is compiled with the OpenMP switch.
  - You can use this to conditionally compile code so that it works with and without OpenMP enabled.

- If you want to link dummy OpenMP library routines into sequential code, there is code in the standard you can copy (Appendix A in 4.0)

Parallel regions

- The overhead of executing a parallel region is typically in the tens of microseconds range
  - depends on compiler, hardware, no. of threads
- The sequential execution time of a section of code has to be several times this to make it worthwhile parallelising.
- If a code section is only sometimes long enough, use the `if` clause to decide at runtime whether to go parallel or not.
  - Overhead on one thread is typically much smaller (<1µs).
- You can use the EPCC OpenMP microbenchmarks to do detailed measurements of overheads on your system.
- Download from **`www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking`**

Is my loop parallelisable?

- Quick and dirty test for whether the iterations of a loop are independent.
- Run the loop in reverse order!!
- Not infallible, but counterexamples are quite hard to construct.

Loops and nowait

```
#pragma omp parallel
{
#pragma omp for schedule(static) nowait
   for(i=0;i<N;i++){
     a[i] = ....
   }
#pragma omp for schedule(static)
   for(i=0;i<N;i++){
     ... = a[i]
   }
}
```

- This is safe so long as the number of iterations in the two loops and the schedules are the same (must be static, but you can specify a chunksize)
- Guaranteed to get same mapping of iterations to threads.

Default schedule

- Note that the default schedule for loops with no schedule clause is implementation defined.
- Doesn’t have to be STATIC.
- In practice, in all implementations I know of, it is.
- Nevertheless you should not rely on this!
- Also note that SCHEDULE(STATIC) does not completely specify the distribution of loop iterations.
  - don’t write code that relies on a particular mapping of iterations to threads

Tuning the chunksize

- Tuning the chunksize for static or dynamic schedules can be tricky because the optimal chunksize can depend quite strongly on the number of threads.

- It’s often more robust to tune the *number of chunks per thread* and derive the chunksize from that.
  - chunksize expression does not have to be a compile-time constant

SINGLE or MASTER?

- Both constructs cause a code block to be executed by one thread only, while the others skip it: which should you use?
- MASTER has lower overhead (it’s just a test, whereas SINGLE requires some synchronisation).
- But beware that MASTER has no implied barrier!
- If you expect some threads to arrive before others, use SINGLE, otherwise use MASTER

Fortran 90 array syntax

- Can’t use loop directives directly to parallelise Fortran 90 array syntax
- WORKSHARE is a worksharing directive (!) which allows parallelisation of Fortran 90 array operations, WHERE and FORALL constructs.
- In simple cases, avoids having to expose a loop.
- Performance portability may be an issue, however!

- Syntax:

```
!$OMP WORKSHARE
        block
!$OMP END WORKSHARE [NOWAIT]
```

Data sharing attributes

- Don’t forget that private variables are uninitialised on entry to parallel regions!

- Can use **`firstprivate`**, but it’s more likely to be an error.
  - use cases for firstprivate are surprisingly rare.

Default(none)

- The default behaviour for parallel regions and worksharing construct is **`default(shared)`**
- This is extremely dangerous - makes it far too easily to accidentally share variables.
- Possibly the worst design decision in the history of OpenMP!
- Always, always use **`default(none)`**
  - I mean always. No exceptions!
  - Everybody suffers from “variable blindness”.

Spot the bug!

```
#pragma omp parallel for private(temp)
   for(i=0;i<N;i++){
      for (j=0;j<M;j++){
         temp = b[i]*c[j];
         a[i][j] = temp * temp + d[i];
      }
   }
```

- May always get the right result with sufficient compiler optimisation!

Private global variables

```
double foo;

#pragma omp parallel \
private(foo)
{
   foo = ....
   a = somefunc();
}
```

```
extern double foo;

double sumfunc(void){
    ... = foo;
}
```

- Unspecified whether the reference to `foo` in `somefunc` is to the original storage or the private copy.
- Unportable and therefore unusable!
- If you want access to the private copy, pass it through the argument list (or use `threadprivate`).

Huge long loops

- What should I do in this situation? (typical old-fashioned Fortran style)

```
do i=1,n
..... several pages of code referencing 100+
    variables
end do
```

- Determining the correct scope (private/shared/reduction) for all those variables is tedious, error prone and difficult to test adequately.

- Refactor sequential code to

```
do i=1,n
   call loopbody(......)
end do
```

- Make all loop temporary variables local to loopbody
- Pass the rest through argument list
- Much easier to test for correctness!
- Then parallelise......
- C/C++ programmers can declare temporaries in the scope of the loop body.

Reduction race trap

```
#pragma omp parallel shared(sum, b)
{
  sum = 0.0;
#pragma omp for reduction(+:sum)
  for(i=0;i<n:i++) {
    sum += b[i];
  }
.... = sum;
}
```

- There is a race between the initialisation of `sum` and the updates to it at the end of the loop.

Missing SAVE or static

- Compiling my sequential code with the OpenMP flag caused it to break: what happened?
- You may have a bug in your code which is assuming that the contents of a local variable are preserved between function calls.
  - compiling with OpenMP flag forces all local variables to be stack allocated and not heap allocated
  - might also cause stack overflow
- Need to use SAVE or static correctly
  - but these variables are then shared by default
  - may need to make them threadprivate
  - “first time through” code may need refactoring (e.g. execute it before the parallel region)

Stack size

- If you have large private data structures, it is possible to run out of stack space.
- The size of thread stack *apart from the master thread* can be controlled by the `OMP_STACKSIZE` environment variable.
- The size of the master thread's stack is controlled in the same way as for sequential program (e.g. compiler switch or using `ulimit`).
  - OpenMP can't control this as by the time the runtime is called it's too late!

Critical and atomic

- You can’t protect updates to shared variables in one place with atomic and another with critical, if they might contend.
- No mutual exclusion between these
  - critical protects code, atomic protects memory locations.

```
#pragma omp parallel
{
#pragma omp critical
   a+=2;
#pragma omp atomic
   a+=3;
}
```

Allocating storage based on number of threads

- Sometimes you want to allocate some storage whose size is determined by the number of threads.
  - but how do you know how many threads the next parallel region will use?
- Can call **`omp_get_max_threads()`** which returns the value of the *nthreads-var* ICV. The number of threads used for the next parallel region will not exceed this
  - except if a **`num_threads`** clause is used.
- Note that the implementation can always deliver fewer threads than this value
  - if your code depends on there actually being a certain number of threads, you should always call **`omp_get_num_threads()`** to check

Using tasks

- Getting the data attribute scoping right can be quite tricky
  - default scoping rules different from other constructs
  - as ever, using `default(none)` is a good idea
- Don’t use tasks for things already well supported by OpenMP
  - e.g. standard do/for loops
  - the overhead of using tasks is greater
- Don’t expect miracles from the runtime
  - best results usually obtained where the user controls the number and granularity of tasks

Environment for performance

- There are some environment variables you should set to maximise performance.
  - don't rely on the defaults for these!

`OMP_WAIT_POLICY=active`

- Encourages idle threads to spin rather than sleep

`OMP_DYNAMIC=false`

- Don't let the runtime deliver fewer threads than you asked for

`OMP_PROC_BIND=true`

- Prevents threads migrating between cores

Debugging tools

- Traditional debuggers such as DDT or Totalview have support for OpenMP
- This is good, but they are not much help for tracking down race conditions
  - debugger changes the timing of event on different threads
- Race detection tools work in a different way
  - capture all the memory accesses during a run, then analyse this data for races which *might have* occured.
  - Intel Inspector XE
  - Oracle Solaris Studio Thread Analyzer

Timers

- Make sure your timer actually does measure wall clock time!
- Do use `omp_get_wtime()` !
- Don’t use `clock()` for example
  - measures CPU time accumulated across all threads
  - no wonder you don’t see any speedup......