Threaded Programming

Lecture 12: OpenMP implementation

epcc

THE UNIVERSITY OF EDINBURGH

Introduction

- How is an OpenMP program actually implemented?
- As a programmer, it may help to understand this
  - understand program performance.
  - write more efficient code.
- We will look at general structure and issues, rather than at a particular implementation.
- A typical OpenMP implementation has two parts: The *compiler* and the *runtime library.*

Compiler

- We won't go into detail about how compilers work...
- An OpenMP compiler transforms code with OpenMP directives to standard code (Fortran/C/C++) with calls to the OpenMP runtime library.
- Two alternatives:
  1. Source-to-source translator
  2. Integral part of f90/cc/CC

Compilers (cont.)

- Source-to-source translator does the transform literally: its output is real, compilable source code with calls to runtime library.
- This is then compiled and linked by a standard compiler.

Pros: portable solution: same compiler can be used on multiple platforms.

Cons: difficult to take advantage of all optimisation opportunities

Compilers (cont.)

- If OpenMP is built in to the standard compiler (most modern implementations), then no transformed source code is produced.

Pros: can better exploit opportunities for optimisation

- can utilise special assembler instructions
- fuller integration with sequential optimiser
- better integration with debuggers.

Cons: non-portable, platform specific solution

Parallel regions

- The body of the parallel region is placed inside a new subroutine.
  - This is called *outlining* (opposite of inlining!)

- The parallel region is replaced by a call to an OpenMP runtime library function (**`run_in_parallel()`**).

- The address of the outlined subroutine is passed as an argument to **`run_in_parallel()`**

Data attribute scoping

- Shared variables are passed in the argument list of the outlined subroutine.
- Private variables are declared locally inside the outlined subroutine.
- Reduction variables require both: a private variable for the local copies and a shared variable for the final result.
- Threadprivate global variables are more awkward.
  - can be implemented using an array of variables with lookup based on thread number
  - need to modify references to the variable.
  - or by dirty tricks in the linker.....

Example

OpenMP source code:

```
      INTEGER MYID, N

!$OMP PARALLEL SHARED(N), PRIVATE(MYID)
      MYID = OMP_GET_THREAD_NUM()
      PRINT *, "Hello from thread ", MYID, " of ", N
!$OMP END PARALLEL
```

Example (cont.)

Transformed code:

```
INTEGER MYID, N

CALL RUN_IN_PARALLEL(_OMP_$1$_PR_,N,....)


SUBROUTINE _OMP_$1$_PR_(N)
INTEGER N                          !SHARED
INTEGER MYID                       !PRIVATE

MYID = OMP_GET_THREAD_NUM()
PRINT *, "Hello from thread ", MYID, " of ", N
END
```

Master and workers

- Master thread executes sequentially until first call to **`run_in_parallel()`**.
- The first time **`run_in_parallel()`** is called, the master thread creates worker threads.
- Master thread assigns task to be done by workers, then also executes task itself.
- Master and workers synchronise at a barrier.
- Master returns from **`run_in_parallel()`** and continues executing sequentially.
- Workers busy wait until master calls **`run_in_parallel()`** again.

Master thread

```
run_in_parallel(task,args)
{
  if (firsttime) {
     for (i=1; i<nthreads; i++)
         pthread_create (&tid, attr, worker_func);
  }
  set_worker_task(task,args);
  task(args);
  barrier();
}
```

Worker threads

```
worker_func()
{
    while(1) {
      wait_for_task();
      task(args);
      barrier();
    }
}
```

Parallel loops

- These are handled in a similar way to parallel regions
- In the outlined subroutine, the real loop bounds are replaced with dummy loop bounds, passed as arguments.
- The runtime library will call the outlined routine for every loop chunk, passing in the required bounds, depending on the chosen schedule.

Example

OpenMP code:

```
!$OMP DO
    DO I = 1,N
       A(I) = B(I) + C(I)
    ENDDO
```

Outlined routine:

```
    SUBROUTINE _OMP_$23$_DO_(A,B,C,START,END)
    INTEGER I
    DO I = START,END
       A(I) = B(I) + C(I)
    ENDDO
```

Synchronisation

- Lock routines can be implemented using Pthread mutexes, or more efficiently via assembly instructions (atomic test-and-set)

- Critical sections are simply a lock/unlock pair.
  - use different locks for differently named sections
  - necessary to manage a global name space of named sections

- Atomic directive can be implemented as a critical section with a special name.
  - but much better to use assembly instructions where available

Barriers

- A simple barrier can be constructed using a locked counter:

```
localsense = !localsense;
lock();
count++;
if (count == NUMTHREADS) {
    count = 0;
    globalsense = localsense;
    unlock();
} else {
    unlock();
    while (globalsense != localsense);
}
```

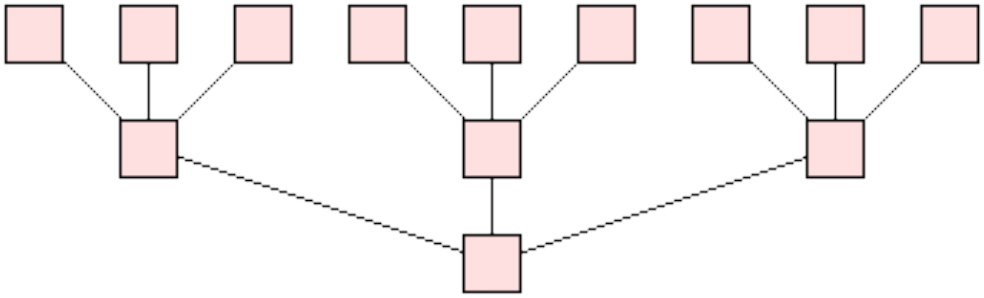
- Note use of sense reversal to avoid re-initialisation problem
- Scales as O(p) – or worse due to contention

Barriers (cont.)

- Barriers can be implemented much more efficiently to using a tree structure and no locks.
  - rely on atomicity of reads/writes
  - scale as O(log P)
  - several possible algorithms, e.g. dissemination barrier, f-way tournament
  - which is best for given architecture depends on details of atomicity guarantees and coherence protocol
  - also use sense reversal to avoid the re-initialisation problem

Dynamic *f*-way tournament

- Analogy is with a knock-out tournament (e.g. tennis)
- *f* threads "compete" in each round
  - a thread notifies arrival by setting a flag
  - checks flags for other threads in their round
  - last to arrive is the "winner": goes on to next round
  - "losers" drop out and spin on a global flag
  - "winner" of the last round is the "champion": toggles the global flag
- *f* = 3 or 4 is usually better than *f* = 2 (fewer rounds)



- Master directive is trivial:

```
if (omp_get_thread_num() = 0)
   {
   }
```

- Single directive is more tricky
  - When a thread arrives it checks a flag. If it is the first to arrive, it sets the flag and executes the block. Otherwise skip the block. Flag requires a mutex: can be a bottleneck.
- Synchronisation points must also implement the implied flush operations
  - via memory fence instructions

Reductions

- Simplest way is to reduce into the shared variable, protected by a mutex lock.
  - inefficient: scales as *O(p)* or worse.
  - causes non-reproducible results for floating point operations. (running identical code on the same number of threads may give different answers on different runs!)

- Better to use a tree structure, similar to a barrier
  - scales as *O(log(p))*
  - can be made reproducible by enforcing the order of operations.
  - array reductions can be parallelised across the array elements

Tasks

- As with other constructs, the task body is outlined into a subroutines.
  - need to store copies of firstprivate variables created when the task was encountered to be used when the task is executed
- Lots of freedom for implementation to choose scheduling policy
  - typically per-thread queues with ability to steal work from other queues
  - choice of LIFO vs FIFO
  - limit on size of queues: when full the current task will be suspended and the thread will execute previously queued tasks
  - trade-off between load balance and locality
  - for locality it is best to execute tasks recently generated by the current thread, but this can restrict parallelism unless the parent task can be resumed on another thread (not the default in OpenMP).

Reusing this material