332 Advanced Computer Architecture

Unassessed tutorial exercise 9

Background: OpenMP

OpenMP (www.openmp.org) is an emerging standard syntax for writing parallel loop programs for shared-memory machines, and is supported by optimising compilers from most major vendors. It is also used as a *target* for compilers which attempt automatic parallelisation.

Here is an extract from a Jacobi solve which should give the idea:

```
!$omp parallel

* Copy new solution into old
!$omp do
         do j=1,m
            do i=1,n
               uold(i,j) = u(i,j)
            enddo
         enddo

* Compute stencil, residual, & update

!$omp do private(resid) reduction(+:error)
         do j = 2,m-1
            do i = 2,n-1
*     Evaluate residual
               resid = (ax*(uold(i-1,j) + uold(i+1,j))
     &                 + ay*(uold(i,j-1) + uold(i,j+1))
     &                  + b * uold(i,j) - f(i,j))/b
* Update solution
               u(i,j) = uold(i,j) - omega * resid
* Accumulate residual error
               error = error + resid*resid
            end do
         enddo
!$omp enddo nowait
!$omp end parallel
```

(this is an extract from `http://www.openmp.org/index.cgi?samples+samples/jacobi.html`). Explanation:

- This is a Fortran program. Lines with a non-space in column 1 are comments. The `&`s mark continuation lines.

- OpenMP takes the form of structured comments - each line starting `!$omp` is an OpenMP directive. The idea is that if the compiler ignores them, the program should still work.
- The `parallel`..`end parallel` directives marks a parallel region of the code for which multiple parallel threads are recruited.
- Each `!$omp do` directive marks a parallel loop. The iterations will be divided into chunks and allocated to available processors until all the iterations are done. The run-time system will try to choose reasonably large chunks to minimise management overheads, and will try to ensure that each processor executes the same group of iterations in successive loops (though the need to balance load often prevents this).
- The directive

  ```
  !$omp do private(resid) reduction(+:error)
  ```

  tells the compiler that the loop can be executed in parallel, but that
  - each processor should have its own copy of the variable `resid`, and
  - the variable `error` is to be summed across all the processors. The implementation can perform the summation in any order.
- The `enddo nowait` directive is a performance optimisation; normally, the implementation inserts a barrier synchronisation at then end of each loop (so the code which follows can assume all iterations have finished). This directive says this synchronisation can be omitted.

Exercise 9.1. Update vs. Invalidate

The performance differences for write-invalidate and write-update schemes can arise from both bandwidth consumption and latency. Assume a memory system with 64-byte cache blocks.

1. Write two parallel loops operating on an array (using elementary OpenMP syntax) to illustrate the bandwidth differences between update and invalidate schemes. One loop should make update look much better, the other should make invalidate look much better.

Exercise 9.2. Degrees of consistency

Consider the following program fragment:

```
/* processor 1 */
A = 0;
⋮
A = 1;
if (B == 0) {
  P();
}
```

```
/* processor 2 */
B = 0;
⋮
B = 1;
if (A == 0) {
  P();
}
```

Assume that procedure P is not called from elsewhere.

1. Assume that the program is running on an idealised shared-memory machine.

   Can both processors execute P simultaneously? Explain your answer carefully.

2. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an invalidation protocol.

   Can both processors execute P simultaneously? Explain your answer carefully.

   What precautions are needed to preserve the expected behaviour?

3. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an *update* protocol.

   Can both processors execute P simultaneously? Explain your answer carefully.

   What precautions are needed to preserve the expected behaviour?

4. Some multiprocessors do not guarantee the expected behaviour for this example, but instead guarantee consistency only when CPUs synchronise (e.g. after passing through a LOCK or BARRIER). Explain why this might offer a performance advantage.

Paul Kelly, Imperial College, 2007

Notes on the solutions to Exercise 9

Exercise 9.1. Update vs. Invalidate

The performance differences for write-invalidate and write-update schemes can arise from both bandwidth consumption and latency. Assume a memory system with 64-byte cache blocks.

1. Write two parallel loops operating on an array (using elementary OpenMP syntax) to illustrate the bandwidth differences between update and invalidate schemes. One loop should make update look much better, the other should make invalidate look much better. **Answer:** There are many ways to go about this. To make update look good, you need one processor to hold a cached copy of data which is written by another processor - and to need the fresh data right away. Try this:

```
      do it=1,niters
!$omp parallel
!$omp do
        do i=1,m
          do j=1,n
            A(i,j) = A(i,j)*2
          enddo
        enddo
!$omp do
        do i=1,m
          do j=1,n
            B(i,j) = A(j,i)+B(i,j)
          enddo
        enddo
!$omp end parallel
      enddo
```

   Assume that each processor executes the same subset of `i` iterations at each `it` iteration. The second loop uses `A` in transposed layout. In the first loop each processor generates a column `A(i,1:n)`. In the second loop, each processor uses a word from each column, thus receiving data from all the processors.

   To construct an example for which invalidate is the better policy, we need to modify the program above so that several assignments are made to each element of `A` before the data is read by another processor. How about:

```
      do it=1,niters
!$omp parallel
        do it2=1,3
!$omp     do
          do i=1,m
            do j=1,n
```

```
            A(i,j) = A(i,j)*2
          enddo
        enddo
      enddo
!$omp do
        do i=1,m
          do j=1,n
            B(i,j) = A(j,i)+B(i,j)
          enddo
        enddo
!$omp end parallel
      enddo
```

I have added a loop around the first loop nest, repating it three times. Note that these three iterations are executed one after the other: the `i` loops remain the only parallel loops.

Exercise 9.2. Degrees of consistency

Consider the following program fragment:

```
/* processor 1 */
A = 0;
:
A = 1;
if (B == 0) {
  P();
}
```

```
/* processor 2 */
B = 0;
:
B = 1;
if (A == 0) {
  P();
}
```

Assume that procedure `P` is not called from elsewhere.

1. Assume that the program is running on an idealised shared-memory machine.

   Can both processors execute `P` simultaneously? Explain your answer carefully. **Answer:** No. If processor 2 reaches its `if` first, it must have executed `B=1` – so when processor 2 gets to its `if` it will find `B!=0`. Ditto if processor 1 is first. In the unlikely event that the processors are in step with one another precisely, it is possible that neither of them executes `P`.

   This reasoning relies on the idea of "strong sequential" memory consistency - that execution is a serial interleaving of the two processors operations consistent with the programorder of them both.

2. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an invalidation protocol.

   Can both processors execute `P` simultaneously? Explain your answer carefully.

What precautions are needed to preserve the expected behaviour? **Answer:** Yes, both processors can execute `P` simultaneously, if you are not careful. Sending the invalidation message is not enough - you need to make sure the invalidation has been done before proceeding.

You could say that the test read must stall until all invalidates/updates on their way have arrived. This isn't very helpful. Smarter is to focus on when the instruction after the write can proceed: it must wait til the invalidate has been acknowledged. But why is this what counts? It depends on a careful examination of the logic behind the program's behaviour. If the write and the read were executed/effected out of order, the mutual exclusion would fail.

3. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an *update* protocol.

   Can both processors execute `P` simultaneously? Explain your answer carefully.

   What precautions are needed to preserve the expected behaviour? **Answer:** This is a bit of a trick question - the answer is the same as above.

4. Some multiprocessors do not guarantee the expected behaviour for this example, but instead guarantee consistency only when CPUs synchronise (e.g. after passing through a LOCK or BARRIER). Explain why this might offer a performance advantage. **Answer:** Can overlap computation with waiting for ack of update/invalidate.

(Paul Kelly, Imperial College, 2007)