HPCSE - I

«Exercise 4 - Additional Feedback on OpenMP»

Panos Hadjidoukas

Question 1 (Roofline)

- Integer operations are not counted
- State clearly your assumptions
- Just try to be careful

Question 2 (Bug hunting)

```
1  #define N 1000
2
3  extern struct data member[N];   // array of structures, defined elsewhere
4  extern int is_good(int i); // returns 1 if member[i] is "good", 0 otherwise
5
6  int good_members[N];
7  int pos = 0;
8
9  void find_good_members()
10 {
11     #pragma omp parallel for
12     for (int i=0; i<N; i++) {
13         if (is_good(i)) {
14             good_members[pos] = i;
15
16             #pragma omp atomic
17             pos++;
18         }
19     }
20 }
```

You had to identify and fix the race condition

Your solution code must

- resolve the issue correctly
- not have compilation issues (typos are expected if written on paper)
- not destroy the parallel performance

Question 2 - I

```
void find_good_members( )
{
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    #pragma omp critical
    if (is_good(i)) {
      good_members[pos] = i;
      pos++;
    }
  }
}
```

There is no race condition but the critical section serializes the parallel loop - there is no exploitation of parallelism.

Question 2 - II

```
void find_good_members( )
{
   #pragma omp parallel for
   for (int i = 0; i < N; i++) {
    if (is_good(i)) {
      #pragma omp critical
      {
        good_members[pos] = i;
        pos++;
      }
   }
}
```

- There is no race condition and the call to is_good() has not been mistakenly included in the critical section.
- A minor issue is that the critical section includes the update of the good_member array. This is avoided in the proposed solution.
- Keep your critical sections as short as possible!

Question 2 - III

```
void find_good_members( )
{
     #pragma omp parallel for
     for (int i = 0; i < N; i++) {
          if (is_good(i)) {
             #pragma omp atomic
             {
               good_members[pos] = i ;
               pos++;
             }
          }
     }
}
```

"pragma omp atomic" can be followed only a valid expression statement. Therefore, the above OpenMP code is not valid.

Question 2 - III

```
void find_good_members( )
{
     #pragma omp parallel for
     for (int i = 0; i < N; i++) {
          if (is_good(i)) {
             #pragma omp atomic capture
             {
               good_members[pos] = i ;
               pos++;
             }
          }
     }
}
```

"pragma omp atomic capture" can be followed by structured blocks of specific form (see next slide).

However, the code in the above example does not comply with the OpenMP rules.

More on "atomic capture"

expression statement	structured block
`v = x++;`	`{v = x; x binop = expr;}`
`v = x--;`	`{v = x; xOP;}`
`v = ++x;`	`{v = x; OPx;}`
`v = --x;`	`{x binop = expr; v = x;}`
`v = x binop = expr;`	`{xOP; v = x;}`
	`{OPx; v = x;}`
	`{v = x; x = x binop expr;}`
	`{x = x binop expr; v = x;}`

`x, v`: are both lvalue expressions with scalar type.
`expr`: is an expression of scalar type that does not reference x.
`binop`: is one of the following binary operators: `+ * - / & ^ | << >>`
`OP`: is one of ++ or --

General advice: whenever possible, keep your code simple!

Reference and examples:
https://www.ibm.com/support/knowledgecenter/SS2LWA_12.1.0/com.ibm.xlcpp121.bg.doc/compiler_ref/prag_omp_atomic.html

Question 3 (dynamic loop scheduling)

```
#pragma omp parallel for schedule(dynamic,1)
for (int i = 0; i < N; i++)
   A[i] = work(i);
```

Possible solution

```
int gi = 0;  // loop-index
#pragma omp parallel
{
  int i; // private value of the loop-index
  while (1) {
    // omp atomic capture followed by i = gi++;
    #pragma omp critical(ompfor)
    {
      i = gi;
      gi++;
    }

    if (i >= N) break;    // necessary check
    A[i] = work(i);       // actual work
  }
}
```

Question 3 - I

```
omp_set_num_threads(N);
#pragma omp parallel
{
  int threadNum = omp_get_thread_num();
  A[threadNum] = work[threadNum];
}
```

Many issues:

- The above code is equivalent to "omp for schedule(static)"
- Missing omp_set_dynamic(0): the number of threads can be < N
- The number of OpenMP threads is limited by the operating system*

(*) https://gitlab.ethz.ch/hpcse17/hs2017/blob/master/examples/openmp1/basic/parallel_maxthreads.c

Question 3 - II

```
int i = 0 ;
#pragma omp parallel
  {
      int myI;
      while (i < N)
      {
          #pragma omp critical
          myI = i++;

          A[myI] = work(myI);
      }
  }
```

Consider the case where all threads see at the same time that variable i equals N-1. Then all of them will proceed to the critical section and the value of the private variable myI will become >= N. This means that the code is exceeding the bounds of the array.

Question 3 - III

```
int i = 0;
#pragma omp parallel
{
  int local_i;

  while (i < N) {
    #pragma omp atomic capture
    local_i = i++;

    a[local_i] = work(local_i);

  }
}
```

Same issues as before (II)

Question 3 - IV

```
    int j = 0;
#pragma omp parallel
    {
        for (int i = 0; i < N;) {
#pragma omp atomic capture
            i = j++;

            if (i < N) A[i] = work(i);
        }
    }
```

No race condition, check of the bounds has been included, minimal number of increments of the private loop index i.

Question 3 - V

```
int i;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        A[i] = work(i);
    }
    ++i;
}
```

- Use of work sharing constructs (such as "omp task") was not allowed in this exercise.
- OpenMP tasks will not be studied this semester.
- The code should be as follows:

```
for(int i = 0; i < N; i++)
   #pragma omp task shared(A) firstprivate(i)
```

Question 3 - VI

```
#pragma omp parallel
{
    int capture;

    while(counter < N) {
        #pragma omp atomic capture
        {capture = counter; counter++;}

        A[capture] = work(capture);
    }
}
```

- Correct structured block for atomic capture
- The code exceeds the bounds of array A

Question 3 - VII

```
count = 0;
#pragma omp parallel
{
  while (count<N)
  {
       #pragma omp atomic
       count++;

       a[count] = work(count);
  }
}
```

- Race condition on shared variable count
- The code exceeds the bounds of array A

Question 3 - VIII

```
#pragma omp parallel
{
    for (int i=0; i<N; i++) {
        #pragma omp single nowait
        {
            a[i] = work(i);
        }
    }
}
```

- Use of work sharing construct (omp single)
- Each thread increases the loop index N times
- Possible increased runtime overheads due to synchronization imposed by the implementation of "omp single"

Question 4 (reduction)

- Straightforward solution based on the idea of
  - splitting "omp parallel" and "omp for"
  - using local (private) variables
  - applying a manual reduction before the end of the parallel region

Question 4 - I

```
double max_rho;
int max_i,  max_j;

max_rho = rho_[0];
max_i = 0;
max_j = 0;

#pragma omp parallel for
for (int j = 0; j < N * N; ++j)
{
     if (rho_[j] > max_rho)
     {
          #pragma omp critical
          {
               max_rho = rho_[j];
               max_j = j;
          }
     }
}
max_i= max_j / N;
max_j= max_j % N;
```

nice manual loop collapse

race condition: multiple threads can find the condition true at the same moment and then update max_rho

critical section within the loop: unnecessary synchronization overhead

Question 4 - II (1/2)

```
double max_rho[NUM_THREADS];
int max_i[NUM_THREADS], max_j[NUM_THREADS];
omp_set_num_threads(NUM_THREADS);

#pragma omp parallel
{
    int threadNum = omp_get_thread_num();
    max_rho[threadNum]=rho_[0];
    max_i[threadNum]=0;
    max_j[threadNum]=0;

    #pragma omp for collapse(2)
    for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
    {
        if (rho_[i*N + j] > max_rho[threadNum])
        {
            max_rho[threadNum] = rho_[i*N + j];
            max_i[threadNum] = i;
            max_j[threadNum] = j;
        }
    }
```

correct code but with increased memory accesses due to false sharing

Question 4 - II (2/2)

```
    #pragma omp master
    {
        int maxRhoThread = 0;
        double max_rhoF = max_rho[0];
        for (int k = 1; k < NUM_THREADS; ++k)
        {
            if (max_rho[k] > max_rhoF)
            {
                max_rhoF = max_rho[k];
                maxRhoThread = k;
            }
        }
        printf("====================================\n");
        printf("Output of compute_max_omp_density():\n");
        printf("Max rho: %.16f\n", max_rho[maxRhoThread]);
        printf("Matrix location: %d %d\n", max_i[maxRhoThread],
        max_j[maxRhoThread]);
    }

}
```

Summary

- Be aware of both correctness and performance mistakes

- Study the example codes available at our gitlab repository
- Become familiar with the OpenMP quick reference card
  - http://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf
- Keep your code simple!

- Avoid race conditions by using synchronization constructs
- Use atomic instead of critical if possible
- The critical regions must include as little code as possible
- Use local (private) variables to avoid false sharing and synchronization

- Good read: M. Süß, C. Leopold, "*Common Mistakes in OpenMP and How to Avoid Them*", 2006.
  - Section 5: OpenMP Programmers Checklist
  - http://www.cse-lab.ethz.ch/images/teaching/HPCSE17/suess06.pdf