OpenMP

(Part II)

Parallel loops are the most frequently used constructs for scientific computing in the shared-memory programming model.

In this chapter we will discuss omp parallel loops.

We begin with the definition of race.

Races

We say that there is a race when there are two memory references taking place in two different tasks such that

1. They are not ordered
2. They refer to the same memory location
3. One of them is a memory write (store).

For example, in the following code there is a race due to the two accesses to a :

```
#pragma omp sections
  {
      #pragma omp section
                {
                ...
                a = x + 5;
                ...
                }
      #pragma omp section
                {
                ...
                y = a + 1;
                ...
                }
   }
```

Another example of a race is:

```
#pragma omp parallel
        {
            ...
            if (omp_get_thread_num()== 0) a=x+5;
            ...
            if (omp_get_thread_num()== 1) a=y+1;
            ...
        }
```

However, there is no race in the following code because the two references to a are ordered by the barrier.

```
#pragma omp parallel
            {
            ...
            if (omp_get_thread_num()==0) a=x+5;
            ...
            #pragma omp barrier
            ...
            if (omp_get_thread_num()==1) a=y+1;
            ...
            }
```

Yet another example of a race is:

```
#pragma omp parallel for
          for (i=0; i<n; i++){
              ...
              a = x[i] + 1;
              ...
          }
```

Here, a is written in all iterations. There is a race if there are at least two threads in the team executing this loop.

And another example is:

```
#pragma omp parallel for
          for (i=1; i<n; i++){
              ...
              a[i] = a[i-1] + 1;
              ...
          }
```

Here, if at least two tasks cooperate in the execution of the loop, some pair of consecutive (say iterations `m` and `m+1`) iterations will be executed by different tasks.

Then, one of the iterations will write to an array element (say `a[m]` in iteration `m`) and the other will read the same element in the next iteration.

Sometimes it is desirable to write a parallel program with races. But most often it is best to avoid races.

In particular, unintentional races may lead to difficult to detect bugs.

Thus, if `a` has the value `1` and `x` the value `3` before the following parallel section starts, `y` could be assigned either `2` or `9`. This would be a bug if the programmer wanted `y` to get the value `9`. And the bug could be very difficult to detect if, for example, `y` were to get the value `2` very infrequently.

```
#pragma omp sections
  {
      #pragma omp section
                {
                ...
                a = x + 5;
                ...
                }
      #pragma omp section
                {
                ...
                y = a + 1;
                ...
                }
   }
```

Race-free parallel loops

Next, we present several forms of parallel loops. In each case, a conventional (sequential) version of the loop will be presented first.

This does not mean that parallel loops can be written only by starting with a conventional loop. However, the most important forms of parallel loops can be easily understood when presented in the context of conventional loops.

The first form of parallel loop can be obtained quite simply. A conventional loop can be transformed into parallel form by just adding a `parallel for` directive if the resulting parallel loop contains no races between any pair of iterations.

An example is the loop

```
for (i=1; i<n; i++){
    ...
    a[i] = b[i] + 1;
    ...
}
```

Notice that this loop computes the vector operation

```
a[1:n]=b[1:n]+1
```

More complex loops can also be directly transformed into parallel form. For example:

```
for (i=1; i<n; i++)
   if (c[i] == 1)
      while (a[i] > eps)
         a[i] = x[i] - x[i-1] / c;
   else
      while (a[i] < upper)
         a[i] = x[i] + y[i+1] * d;
```

Notice that although consecutive iterations access the same element of `x`, there is no race because both accesses are reads.

Privatization

Sometimes the transformation into parallel form requires the identification of what data should be declared as `private`.

For example, consider the following loop:

```
for (i=0; i<n; i++){
    x = a[i]+1 ;
    b[i] = x + 2 ;
    c[i] = x * 2 ;
}
```

This loop would be fully parallel if it were not for `x` which is stored and read in all iterations.

One way to avoid the race is to eliminate the assignment to `x` by forward substituting `a[i]+1` :

```
for (i=0; i<n; i++){
    b[i] = (a[i]+1) + 2
    c[i] = (a[i]+1) * 2
}
```

A simpler way is to declare `x` as private:

```
#pragma omp parallel for private(x)
   for (i=0; i<n; i++){
      x = a[i]+1;
      b[i] = x + 2;
      c[i] = x * 2;
   }
```

In general, a scalar variable can be declared `private` if

1. It is always assigned before it is read in every iteration of the loop, and
2. It is never used again, or it is reassigned before used again after the loop completes.

There is a copy of each private variable for each thread participating in the execution of the parallel for.

If `x` were declared inside the iteration as follows

```
for (i=0; i<n; i++){
      float x;
      x = a[i]+1;...
```

there would be a copy for each iteration

Sometimes it is necessary to privatize arrays. For example, the loop

```
for (i=0; i<n; i++){
    for (j=0; j<n; j++)
        y[j] = a[i][j] + 1);
    ...
    for (k=1; k<n-1; k++)
        b[i][k] = y[k] * 2;
}
```

can be directly parallelized if vector `y` is declared `private`.

An array can be declared `private` if

1. No element of the array is read before it is assigned within the same iteration of the loop.
2. Any array element used after the loop completed is reassigned before it is read.

An important case arises when the variable to be privatized is read after the loop completes without reassignment.

For example

```
for (i=0; i<n; i++){
    x = a[i]+1;
    b[i] = x + 2;
    c[i] = x ** 2;
}

...=x;
```

One way to solve this problem is to "peel off" the last iteration of the loop and then parallelize:

```
#pragma omp parallel for private(x)
   for (i=0; i<n-1; i++){
      x = a[i]+1;
      b(i) = x + 2;
      c(i) = x * 2;
   }
   x=a[n-1]+1;
   b[n]=x+2;
   c[n]=x+2;
```

An equivalent, but simpler approach is to declare `x` as `lastprivate`.

```
#pragma omp parallel for lastprivate(x)
   for (i=0; i<n; i++){
      x = a[i]+1;
      b[i] = x + 2;
      c[i] = x * 2;
   }
```

A similar situation arises when a private variable needs to be initialized with values from before the loop starts execution. Consider the loop:

```
for (i=0; i<n; i++){
    for (j=0; j<n; j++){
        a[j] = calc_a(j);
        b[j] = calc_b(i,j);
    }
    for (j=0; j<n; j++){
        x=a[j]-b[j];
        y=b[j]+a[j];
        c[i][j]=x*y;
    }
}
```

To parallelize this for loop, `x`, `y`, `a` and `b` should be declared private. However, in iteration `i` the value of `a[i+1], a[i+2],...,a[n]` and of `b[i+1], b[i+2],..., b[n]` are those assigned before the loop starts.

To account for this, `a` and `b` should be declared as `firstprivate`.

```
#pragma omp parallel for private(j,x,y), firstprivate(a,b)
```

Induction variables

Induction variables appear often in scientific programs. These are variables that assume the values of an arithmetic sequence across the iterations of the loop:

For example, the loop

```
for (i=0; i<n; i++){
    j += 2;
    for (k=0; k<j; k++)
        a[k][j] = b[k][j] + 1;
}
```

cannot be directly transformed into parallel form because the satement `j+=2` produces a race. And `j` cannot be privatized because it is read before it is assigned.

However, it is usually easy to express induction veriables as a function of the loop index. So, the previous loop can be tranformed into:

```
for (i=0; i<n; i++){
    m = 2*(i+1) + j;
    for (k=0; k<m; k++)
        a[k][m] = b[k][m] + 1;
}
```

In this last loop, `m` can be made private and the loop directly tranformed into parallel form.

If the last value of variable `j` within the loop is used after the loop completes, it is necessary to add the statement

```
j+=2*n;
```

immediately after the loop to set the variable `j` to its correct value.

Most induction variables are quite simple, like the one in the previous example. However, in some cases a more involved formula is necessary to represent the induction variable as a function of the loop index:

For example consider the loop:

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++){
        k+=2;
        a[k]=b[i][j]+1;
}
```

The only obstacle for the parallelization of loop `i` is the induction variable `k`. Notice that no two iterations assign to the same element of array `a` because `k` always increases from one iteration to the next.

The formula for `k` is somewhat more involved than the formula of the previos example, but still is relatively simple:

```
#pragma omp parallel for private(j)
   for (i=0; i<n; i++)
      for (j=0; j<m; j++)
         a[2*(m*i+j+1)+k]=b[i][j]+1;
   k=2*n*m+k;
```

or

```
#pragma omp parallel for private(j,k)
   for (i=0; i<n; i++){
      k=2*m*i;
      for (j=0; j<m; j++){
         k+=2;
         a[k]=b[i][j]+1;
      }
   }
```

As a final example, consider the loop:

```
for (i=0; i<n; i++){
   j+=1;
   a[j]= b[i]+1;
   for (k=0; k<i+1; k++){
      j+=1;
      c[j]=d[i][k]+1;
   }
}
```

Here, again, only the induction variable, `j`, causes problems. But now the formulas are somewhat more complex:

```
#pragma omp parallel for private(k)
   for (i=0; i<n; i++)
      a[(i+1)+i*(i+1)/2]= b[i]+1;
      for (k=0; k<i+1; k++)
         c[i+(i+1)*(i+2)/2+k]=d[i][k]+1;
   j=n+n*(n+1)/2;
```

Sometimes, it is necessary to do some additional transformations to remove induction veriables. Consider the following loop:

```
j=n;
for (i=0; i<n; i++){
   b[i]=(a[j]+a[i])/2.;
   j=i;
}
```

Variable `j` is called a *wraparound* variable of first order. It is called first order because only the first iteration uses a value of `j` from outside the loop. A wraparound variable is an induction variable whose value is carried from one iteration to the next.

The way to remove the races produced by `j` is to peel off one iteration, move the assignment to `j` from one iteration to the top of the next iteration (notice that now `j` must be assigned `i-1`), and then privatize :

```
        j=n;
        if (n>=1) {
           b[1]=(a[j]+a[1])/2.;
#pragma omp parallel for private(i),lastprivate(j)
           for (i=1; i<n; i++){
              j=i-1;
              b[i]=(a[j]+a[i])/2.;
           end do
        }
```

Notice that the if statement is necessary to make sure that the first iteration is executed only if the original loop would have executed it.

Alternatively, the wraparound variable could be an induction variable. The transformation in this case is basically the same as above except that the induction variable has to be expressed in terms of the loop index first.

Consider the loop:

```
j=n;
for (i=0; i<n; i++){
   b[i]=(a[j]+a[i])/2.;
   j+=1;
}
```

As we just said, we first replace the right hand side of the assignment to `j` with an expression involving `i`.

```
j=n;
for (i=0; i<n; i++){
   b[i]=(a[m]+a[i])/2.;
   m=i+j;
}
j+=n;
```

Notice that we changed the name of the variable within the loop to be able to use the value of `j` coming from outside the loop.

We can now proceed as above to obtain:

```
      j=n;
      if (n>=1) {
         b[1]=(a[j]+a[i])/2.;
#pragma omp parallel for private (m)
         for (i=1; i<n; i++){
            m=i-1+j;
            b[i]=(a[m]+a[i])/2.;
         }
         j=n+j; /*this must be inside the if */
      }
```


Critical Regions and Reductions

Consider the following loop:

```
int a[][], sum;
...
for (i=0; i<n; i++)
   for (j=0; j<m; j++){
      a[i][j]=b[i][j]+d[i][j];
      sum+=a[i][j];
   }
```

Here, we have a race due to `sum`. This race cannot be removed by the techniques discussed above. However, the `+` operation used to compute `sum` is associative and `sum` only appears in the statement that computes its value.

The integer addition operation is not really associative, but in practice we can assume it is if the numbers are small enough so there is never any overflow.

Under these circumstances, the loop can be transformed into the following form:

```
#pragma omp parallel private(local_sum)
{
    local_sum=0;
    #pragma omp for nowait
    for (i=0; i<n; i++)
        for (j=0; j<m; j++)
            local_sum += a[j][i];
    #pragma omp critical
    sum+=local_sum;
}
```

Here, we use the critical directive to avoid the following problem.

The statement

```
sum=sum+local_sum
```

will be translated into a machine language sequence similar to the following:

```
load     register_1,sum
load     register_2,local_sum
add      register_3,register_1,register_2
store    register_3,sum
```

Assume now there are two tasks executing the statement

```
sum=sum+local_sum
```

simultaneously. In one `local_sum` is `10`, and in the other `15`. Assume `sum` is `0` when both tasks start executing the statement. Consider the following sequence of events:

As can be seen, interleaving the instructions between the two

time	task 1	sum	task 2
1	load r1,local_sum	0	
2	load r2, sum	0	load r1,local_sum
3	add r3,r2,r1	0	load r2,sum
4	store r3, sum	10	add r3,r2,r1
5		15	store r3,sum

tasks produces incorrect results. The critical directive precludes this interleaving. Only one task at a time can execute a critical region with the same name.

Short Hand Notation for Reduction

The reduction clause can be used to specify various classes of reductions.

```
            int a[][], sum;
            ...
  #pragma omp parallel for private(j) reduction(+:sum)
            for (i=0; i<n; i++)
                for (j=0; j<m; j++){
                    a[i][j]=b[i][j]+d[i][j]
                    sum+=a[i][j]
                }
```

The general form of the clause is `reduction(operator:list)` where operator is:

Operator	Initialization value
+	0
*	1
-	0
&	~0 (all bits 1)
|	0
^	0
&&	1
||	0

Performance considerations

Numerous factors affect the performance of OpenMP programs. These include:

1. Sequential performance. All issues relevant to the performance of sequential codes and in particular locality are of course important for the performance of OpenMp codes. However, redundant computations sometimes improve parallel performance.
2. Overhead. There is a cost associated with initialization and termination of parallel regions, workshare for loops, barriers, and critical sections.
3. Load balancing and granularity. The scheduling strategy influences these.
4. Communication. Although there is no explicit communication, parallel computations always involve communication. In the case of shared-memory programs, attention must be paid to memory accesses and cache behavior

Sequential performance

Locality is of crucial importance not only for sequential but also for parallel performance. The tiling techniques discussed earlier are of great importance here.

Sometimes redundat computation can be useful for paralle peformance. So, in

```
for (i=1; i<n; i++)
            c[i] = 3*i;
#pragma omp parallel
{
        ...
       use of c[]
        ...
}
```

the values of `c` must be tranferred from the master thread to each of the threads executing the parallel region. This would typically involve copying across caches.

Moving the computation of `c` inside the parallel region would avoid this problem. There would generate `n` additional computations inside each thread, but would avoid the data copying.

Overhead

Sometimes it is better to execute serially. Typically this happens when the amount of computation is too small to justify the overhead. In OpenMP, the `if` clause:

```
#pragma omp parallel for if(n>MIN_NUM)
            for (i=1; i<n; i++){
                ...
                a[i] = a[i-1] + 1;
                ...
            }
```

dynamically controls when the poop will be executed in parallel.

Sometimes the best way to identify the minimum number of iterations (`MIN_NUM`)is by experimentally trying several values.

The overhead is also influenced by the granularity.

Load balancing and granularity

The schedule clause is designed to control these factors. Some of the scheduling options are:

`static`

When `schedule(static,` *chunk_size) is specified, iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.*

`dynamic`

When `schedule(dynamic,` *chunk_size) is specified, the iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.*

guided

When **schedule**(**guided**, *chunk_size) is specified, the iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned.*

For a chunk_size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a chunk_size with value k (greater than 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations).

Communication

Issues of communication are of course of great importance. One of the difficulties of OpenMP is that communication is often implicit. This has been mentioned as a disadvantage of OpenMP relative to MPI.

The example mentioned above under sequential performance where `c` was computed redundantly is an example of transformations to reduce communication.

False sharing can cause performance dgradation. In false sharing diiferent threads access different data items collocated in the same cache line. This can cause significant communication costs.

For example, in the program

```
#pragma omp parallel for schedule(dynamic,1)
            for (i=1; i<n; i++){
                ...
                a[i] = ...;
                ...
            }
```

each cahce line may have to travel across several threads to enable the assigment of values to the different elements of array `a`.