OpenMP

(Part 1)

Introduction

OpenMP is a collection of compiler directives, library routines, and environment variables to specify shared memory parallelism.

This collection was designed by committee involving computer vendors including Intel, HP, IBM, and SGI.

Thereare Fortran, C and C++ directives.

See http://www.openmp.org

The `parallel` directive

The `parallel` construct defines a parallel region.

An OpenMP program begins execution as a single thread, called the initial thread. When a thread encounters a parallel construct it becomes a master thread which creates a team of threads. The statements enclosed by the parallel construct, are executed by each thread in the team.

At the end of the parallel construct the threads in the team synchronize and only the master thread continues execution.

The general form of this construct is:

`#pragma omp parallel` [clause[[,]clause] ...] new-line
    structured-block

where clause is one of

- **`if`(scalar-expression)**
- **`num_threads`(integer-expression)**
- **`default(shared|none)`**
- **`private`(list)**
- **`firstprivate`(list)**
- **`shared`(list)**
- **`copyin`(list)**
- **`reduction`(operator: list)**

Next, we discuss the `private` (*list*) clause.

Private variables are undefined upon entering the `parallel` construct and are also undefined on exit from it.

Consider,

```
          c = fun (d);
          for (i=0;i <n; i++) a[i] = b[i] + c;
          for (i=0; i<n; i++) e[i] = a[20]+ a[15];
```

A simple OpenMP implementation is

```
    float c = fun(d);
    int nt = omp_get_num_threads();
#pragma omp parallel private(i,il,iu,tn)

    {   int tn=omp_get_thread_num();
        int il=(i<(n%nt))?((n+nt-1)/nt)*i:(n/nt)*i + n%nt;
        int iu=(i<(n%nt))?((n+nt-1)/nt)*(i+1)-1:(n/nt)*(i+1)+n%nt-1;

        for (i=il; i <= iu; i++)  a[i] = b[i] + c;
    }

    for (i=0; i<n; i++) e[i] = a[20] + a[15];
```

The first two statements can be moved inside the parallel region and `c` and `nt` can be declared private if they are not used again after the loop.

```
#pragma omp parallel private(i,il,iu,tn,c,nt)

    {   float c = fun(d);
        int nt = omp_get_num_threads();

        int tn=omp_get_thread_num();
        int il=(i<(n%nt))?((n+nt-1)/nt)*i:(n/nt)*i + n%nt;
        int iu=(i<(n%nt))?((n+nt-1)/nt)*(i+1)-1:(n/nt)*(i+1)+n%nt-1;

        for (i=il; i <= iu; i++)
            a[i] = b[i] + c;
    }

    for (i=0; i<n; i++) e[i] = a[20] + a[15];
```

The `barrier` directive

To incorporate the assignments to `e[i]` into the parallel region it is necessary to make sure that `a[20]` and `a[15]` have been computed before the for statement executes.

This can be done with a `barrier` construct which synchronizes all the threads in the enclosing `parallel` region. When encountered, each thread waits until all the others in that team have reached this point.

```
#pragma omp parallel private(i,il,iu,tn,c,nt)

    {   float c = fun(d);
        int nt = omp_get_num_threads();
        int tn=omp_get_thread_num();
        int il=(i<(n%nt))?((n+nt-1)/nt)*i:(n/nt)*i + n%nt;
        int iu=(i<(n%nt))?((n+nt-1)/nt)*(i+1)-1:(n/nt)*(i+1)+n%nt-1;

        for (i=il; i <= iu; i++)
            a[i] = b[i] + c;
        #pragma omp barrier
        for (i=il; i<iu; i++) e[i] = a[20] + a[15];
    }
```

The for construct

A simpler way to write the previous code uses the for directive:

```
#pragma omp parallel private(c)
    {   float c = fun(d);
        #pragma omp for
        for (i=0; i < n; i++) a[i] = b[i] + c;
        ~~#pragma omp barrier~~
        #pragma omp for
        for (i=0; i< n; i++) e[i] = a[20] + a[15];
    }
```

The for construct specifies that the iterations of the immediately following for loop will be distributed among the different trehads in the team executing the parallel region.

The pragma omp barrier is not needed since the first for construct generates an implicit barrier.

The syntax of the `for` construct is as follows:

```
#pragma omp for [clause[[,] clause] ... ] new-line
       for-loops
```

There are several `for` *clauses* including `private` and `schedule`.

The schedule could assume other values including `dynamic` .

The `nowait` clause eliminates the implicit barrier at the end of the for loop. In the previous example, the `nowait` clause an be added to both loops because of the explicit barrier at the end of the first for loop and the implicit barrier at the end of the second loop.

Another example of `for` with the `nowait` clause is

```
void forfun(a,b,c,d,m,n)
        int m,n;
        float a[n][n],b[n][n],c[m][m],d[m][m];
        {
        #pragma omp parallel private(j)
            {
            #pragma omp for schedule(dynamic),nowait
            for (i=1; i < n; i++)
                for (j=0; j <i; j++)
                    b[j][i]=(a[j][i]+a[j][i+1])/2;

            #pragma omp for schedule(dynamic),nowait
            for (i=1; i < m; i++)
                for (j=0; j <m; j++)
                    d[i][j]=(c[j][i]+c[j][i-1])/2;
            }
        }
```

In this case it is correct for any thread in the team to proceed to the second loop before the first loop has completed since the two loops operate on completely different arrays.

Question: Would the nowait be correct if we replace `c` by `b` in the second loop ?

The `parallel for` construct

An alternative to the `for` is the `parallel for` construct which is no more than a shortcut for a `parallel` construct containing a single `for` construct.

For example, the following code segment

```
#pragma omp parallel
#pragma omp for schedule(dynamic), nowait
        for (i=1; i < n; i++)
             b[i]=(a[i]+a[i+1])/2;
```

could be rewritten

```
#pragma omp parallel for schedule(dynamic)
        for (i=1; i < n; i++)
             b[i]=(a[i]+a[i+1])/2;
```

And the routine `forfun` can be rewritten as follows:

```
void forfun(a,b,c,d,m,n)
      int m,n;
      float a[n][n],b[n][n],c[m][m],d[m][m];
      {
         int i,j;
   #pragma omp parallel for schedule(dynamic)
         for (i=1; i < n; i++)
            for (j=0; j <i; j++)
               b[j][i]=(a[j][i]+a[j][i+1])/2;

   #pragma omp parallel for schedule(dynamic)
         for (i=1; i < m; i++)
            for (j=0; j <m; j++)
               d[i][j]=(c[j][i]+c[j][i-1])/2;
      }
```

There are two disadvantages to this last version of `forfun`:

1. There is a barrier at the end of the first loop.
2. There are two parallel regions. There is overhead at the beginning of each.

The `single` construct

The `single` construct has the following syntax:

```
#pragma omp single [clause[[,] clause] ...] new-line
        structured-block
```

The enclosed region of code is executed by only one of the tasks in the team.

Tasks in the team not executing the `single` block wait at the `end single`, unless `nowait` is specified.

An example of `single`:

```
 void sp_1a(a,b,n){
#pragma omp parallel private(i)
 {
     #pragma omp for
      for (i=0; i < n; i++) a[i]=1.0/a[i];
        #pragma omp single
         a[1]=min(a[1],1.0);
     #pragma omp for nowait
      for (i=0; i < n; i++) b[i]=b[i]/a[i];
 }
```

The sections construct

An alternative way to write the `forfun` routine is:

```
void forfun(a,b,c,d,m,n)
    int m,n;
    float a[n][n],b[n][n],c[m][m],d[m][m];
    {
    #pragma omp sections private(i,j)
        {
        #pragma omp section
            for (i=1; i < n; i++)
                for (j=0; j <i; j++)
                    b[j][i]=(a[j][i]+a[j][i+1])/2;

        #pragma omp section
            for (i=1; i < m; i++)
                for (j=0; j <m; j++)
                    d[i][j]=(c[j][i]+c[j][i-1])/2;
    }
```

The `sections` directive specifies that the enclosed sections of code are to be divided among threads in the team. Each section is executed by one thread in the team. Its syntax is as follows:

```
#pragma omp sections [clause[[,] clause] ...] new-line
   {
    [#pragma omp section new-line]
         structured-block
    [#pragma omp section new-line
         structured-block  ]
    . . .
    }
```