Chapter 3. OpenMP

3.1 Introduction

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

This collection has been designed with the cooperation of many computer vendors including Intel, HP, IBM, and SGI. So, it is likely to become the standard (and therefore portable) way of programming SMPs.

Fortran and C directives have been defined.

3.2 The PARALLEL directive

The `parallel/end parallel` directive pair defines a parallel region and constitutes as parallel construct.

An OpenMP program begins execution as a single task, called the *master thread*. When a parallel construct is encountered, the master thread creates a team of threads. The statements enclosed by the parallel construct, including routines called from within the enclosed construct, are executed in parallel 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:

```
C$omp parallel [parallel-clause[[,]parallel-clause] ...]
   parallel region
C$omp end parallel
```

There are several classes of parallel-clauses. Next, we discuss the `private(`*list*`)`clause.

All variables are assumed to be shared by all tasks executing the parallel region. However, there will be a separate copy of each variable listed in a `private` clause for each task. There will also be an additional copy of the variable that can be accessed outside the parallel region.

Variables defined as private are undefined for the thread entering the construct and are also undefined for the thread on exit from a parallel construct.

As an example, consider the following code segment

```
      c = sin (d)
      forall i=1 to n
         a(i) = b(i) + c
      end forall
      e = a(20)+ a(15)
```

A simple OpenMP implementation would take the form

```
      c = sin(d)
c$omp parallel private(i,il,iu)
      call get_limits(n,il,iu,
     *                omp_get_num_threads(),
     *                omp_get_thread_num())
      do i=il,iu
         a(i) = b(i) + c
      end do
c$omp end parallel
      e = a(20) + a(15)
```

Notice that the first statement can be incorporated into the parallel region. In fact, c can be declared as private assuming it is never used outside the loop.

```
c$omp   parallel private(c,i,il,iu)
        c= sin(d)
        call get_limits(n,il,iu,
      *                 omp_get_num_threads(),
      *                 omp_get_thread_num())
        do i=il,iu
           a(i) = b(i) + c
        end do
c$omp   end parallel
        e = a(20) + a(15)
```

3.3 The BARRIER directive

To incorporate `e` into the parallel region it is necessary to make sure that `a(20)` and `a(15)` have been computed before the statement executes.

This can be done with a `barrier` directive 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.

```
c$omp   parallel private(c,i,il,iu)
        c = sin(d)
        call get_limits(n,il,iu,
     *                  omp_get_num_threads(),
     *                  omp_get_thread_num())
        do i=il,iu
           a(i) = b(i) + c
        end do
c$omp   barrier
        e = a(20) + a(15)
c$omp   end parallel
```

3.4 The SINGLE directive

Finally, since e is shared, it is not a good idea for all tasks in the team to execute the last assignment statement. There will be several redundant assignments all competing for access to the single memory location. Only one task needs to execute the assignment.

This can be accomplished with the psingle directive:

```
c$omp   parallel private(c,i,il,iu)
        c = sin(d)
        call get_limits(n,il,iu,
     *                  omp_get_num_threads(),
     *                  omp_get_thread_num())
        do i=il,iu
           a(i) = b(i) + c
        end do
c$omp   barrier
c$omp   single
        e = a(20) + a(15)
c$omp   end single nowait
c$omp   end parallel
```

The `single` directive has the following syntax:

```
c$omp   single [single-clause[[,]single-clause] ...]
        block
c$omp   end single [nowait]
```

This directive specifies that the enclosed region of code is to be executed by one and only one of the tasks in the team.

Tasks in the team not executing the `psingle` block wait at the `end psingle`, unless `nowait` is specified. In this case, there is no need for this implicit barrier since one already exists at the `end parallel` directive.

One of the two *single-clauses* is `private(`*list*`).`

A better example of `psingle`:

```
        subroutine sp_1a(a,b,n)
c$omp   parallel private(i)
c$omp      pdo
              do i=1,n
                 a(i)=1.0/a(i)
              end do
c$omp      single
              a(1)=min(a(1),1.0)
c$omp      end single
c$omp      pdo
              do i=1,n
                 b(i)=b(i0/a(i)
c$omp      end pdo nowait
c$omp   end parallel
        end
```

3.5 The DO directive

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

```
c$omp   parallel private(c,i,il,iu)
        c = sin(d)
c$omp   do schedule(static)
        do i=1,n
           a(i) = b(i) + c
        end do
c$omp   end do
c$omp   single
        e = a(20) + a(15)
c$omp   end single nowait
c$omp   end parallel
```

The pdo directive specifies that the iteration s of the immediately following do loop must be executed in parallel.

The syntax of the `do` directive is as follows:

```
c$omp    do [do-clause[[,]do-clause] ...]
         do loop
c$omp    end do [nowait]
```

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

The schedule could assume other values including `dynamic`.

The `nowait` clause eliminates the implicit barrier at the `end do` directive. In the previous example, the `nowait` clause should not be used.

An example of do with the nowait directive is

```
        subroutine pdo_2(a,b,c,d,m,n)
        real a(n,n),b(n,n),c(m,m), d(m,m)
c$omp   parallel private(i,j)
c$omp      do schedule(dynamic)
              do i=2,n
                 do j=1,i
                    b(j,i)=(a(j,i)+a(j,i+1))/2
                 end do
              end do
c$omp      end do nowait
c$omp      do  schedule(dynamic)
              do i=2,m
                 do j=1,i
                    d(i,j)=(c(j,i)+c(j,i-1))/2
                 end do
              end do
c$omp      end do nowait
c$omp   end parallel
        end
```

3.6 The PARALLEL DO directive

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

For example, the following code segment

```
c$omp   parallel private(i)
c$omp      do schedule(dynamic)
              do i=1,n
                 b(i)=(a(i)+a(i+1))/2
              end do
c$omp      end do nowait
c$omp   end parallel
        end
```

could be rewritten

```
c$omp   parallel do
c$omp&  private(i)
c$omp&  schedule(dynamic)
              do i=1,n
                 b(i)=(a(i)+a(i+1))/2
              end do
c$omp   end parallel do
        end
```

And the routine `pdo_2` can be rewritten as follows:

```
      subroutine pdo_2(a,b,c,d,m,n)
      real a(n,n),b(n,n),c(m,m), d(m,m)
c$omp parallel do
c$omp& private(i,j)
c$omp& schedule(dynamic)
            do i=2,n
               do j=1,i
                  b(j,i)=(a(j,i)+a(j,i+1))/2
               end do
            end do
c$omp end parallel do
c$omp parallel do
c$omp& private(i,j)
c$omp& schedule(dynamic)
            do i=2,m
               do j=1,i
                  d(i,j)=(c(j,i)+c(j,i-1))/2
               end do
            end do
c$omp    end parallel do
      end
```

There are two disadvantages to this last version of pdo_2:

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.

3.7 The SECTIONS directive

An alternative way to write the pdo_2 routine is:

```
        subroutine pdo_2(a,b,c,d,m,n)
        real a(n,n),b(n,n),c(m,m), d(m,m)
c$omp   parallel private(i,j)
c$omp      sections
c$omp         section
                 do i=2,n
                    do j=1,i
                       b(j,i)=(a(j,i)+a(j,i+1))/
2
                    end do
                 end do
c$omp         psection
                 do i=2,m
                    do j=1,i
                       d(i,j)=(c(j,i)+c(j,i-1))/
2
                    end do
                 end do
c$omp      end sections nowait
c$omp   end parallel
        end
```

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:

```
c$omp   sections[sections-clause[[,]sections-clause] ...]
[c$omp     section]
              block
[[c$omp    section
              block]
                 .
                 .
                 .]
c$omp   end sections [nowait]
```

Chapter 4. Parallel Loops in OpenMP

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.

4.1 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`:

```
c$omp   parallel sections
c$omp      psection
              ...
              a = x + 5
              ...
c$omp      psection
              ...
              y = a + 1
              ...
c$omp   end parallel sections
```

Another example of a race is:

```
c$omp   parallel
           ...
           if (omp_get_thread_num().eq.0) a=x+5
           ... [no omp directive here]
           if (omp_get_thread_num().eq.1) a=y+1
           ...
c$omp   end parallel
```

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

```
c$omp   parallel
           ...
           if (omp_get_thread_num().eq.0) a=x+5
           ...
c$omp      barrier
           ...
           if (omp_get_thread_num().eq.1) a=y+1
           ...
c$omp   end parallel
```

Another example of a race is:

```
c$omp   parallel do
           do i=1,n
              ...
              a = x(i) + 1
              ...
           end do
c$omp   end parallel do
```

Here, a is written in all iterations. There is a race if there are at least two tasks executing this loop. (It is ok to execute an OpenMP program with a single processor)

Another example is:

```
c$omp   parallel do
           do i=1,n
              ...
              a(i) = a(i-1) + 1
              ...
           end do
```

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

Then, one of the iterations will write to an array element (say `a(j)` in iteration `j`) 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.

```
c$omp   parallel sections
c$omp      section
              ...
              a = x + 5
              ...
c$omp      section
              ...
              y = a + 1
              ...
c$omp   end parallel sections
```

4.2 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 loop` directive if the resulting parallel loop contains no races between any pair of iterations.

An example is the loop

```
do i=1,n
   a(i) = b(i) +1
end do
```

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:

```
      do i=1,n
         if (c(i) .eq. 1) then
            do while (a(i) .gt. eps)
               a(i) = x(i) - x(i-1) / c
            end do
         else
            do while (a(i) .lt. upper)
               a(i) = x(i) + y(i+1) * d
            end do
         end if
      end do
```

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

4.3 Privatization

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

For example, consider the following loop:

```
do i=1,n
   x = a(i)+1
   b(i) = x + 2
   c(i) = x ** 2
end do
```

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`:

```
do i=1,n
   b(i) = (a(i)+1) + 2
   c(i) = (a(i)+1) ** 2
end do
```

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

```
c$omp   parallel do private(i,x)
        do i=1,n
           x = a(i)+1
           b(i) = x + 2
           c(i) = x ** 2
        end do
```

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.

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

```
do i=1,n
   do j=1,n
      y(j) = a(i,j) + 1)
   end do
   ...
   do k=2,n-1
      b(i,j) = y(j) ** 2
   end do
end do
```

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

```
      do i=1,n
         x = a(i)+1
         b(i) = x + 2
         c(i) = x ** 2
      end do

      ...=x
```

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

```
c$omp   parallel do private(i,x)
      do i=1,n-1
         x = a(i)+1
         b(i) = x + 2
         c(i) = x ** 2
      end do
      x=a(n)+1
      b(n)=x+2
      c(n)=x+2
```

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

```
c$omp   parallel do private(i) lastprivate(x)
      do i=1,n
         x = a(i)+1
         b(i) = x + 2
         c(i) = x ** 2
```

```
end do
```

Variables in lastprivate are private variables; however, in addition, at the end of the do loop, the thread that executes the last iteration updates the version of the variable that exists outside the loop.

If the last iteration does not assign a value to the entire variable, the variable is undefined after the loop.

For example, if `c(n) > 5` in the loop:

```
c$omp   parallel do private(i) lastprivate(x)
      do i=1,n
         if (c(i).lt.5) then
            x=b(i)+1
            a(i)=x+x**2
         end if
      end do
```

then `x` would not be defined after the loop.

Similarly, if the private variable is a vector, only the elements assigned in the last iteration will be defined. (in KAI's version, the elements not assigned in the last iteration are 0).

For example, the program:

```
      real a(100),b(100),c(100)
      do i=1,100
         a(i)=1
      end do
      do i=1,100
         b(i)=3
      end do
      print *,a(1),a(2)
      b(1)=1
c$omp parallel do lastprivate(a)
      do i=1,100
         do j=1,100
            if (b(j).lt.3) then
               a(j)=3
               c(j)=a(j)
            end if
         end do
      end do
      print *,a(1),a(2)
      end
```

prints

```
1.000000   1.00000
3.000000   0.
```

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

```
do i=1,n
   do j=1,i
      a(j) = calc_a(j)
      b(j) = calc_b(i,j)
   end do
   do j=1,n
      x=a(j)-b(j)
      y=b(j)+a(j)
      c(i,j)=x*y
   end do
end do
```

To parallelize this 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.

```
c$omp   parallel do private(i,j,x,y)
c$omp&  firstprivate(a,b)
```

4.4 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

```
do i=1,n
   j = j + 2
   do k=1,j
      a(k,j) = b(k,j) + 1
   end do
end do
```

cannot be directly transformed into parallel form because the satement `j=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:

```
do i=1,n
   m=2*i+j
   do k=1,m
      a(k,m) = b(k,m) + 1
   end do
end do
```

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+j
```

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:

```
do i=1,n
   do j=1,m
      k=k+2
      a(k)=b(i,j)+1
   end do
end do
```

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:

```
c$omp   parallel do private(i,j)
```

```
do i=1,n
   do j=1,m
      a(2*(m*(i-1)+j)+k)=b(i,j)+1
   end do
end do
k=2*n*m+k
```

As a final example, consider the loop:

```
do i=1,n
   j=j+1
   a(j)= b(i)+1
   do k=1,i
      j=j+1
      c(j)=d(i,k)+1
   end do
end do
```

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

```
c$omp   parallel do private(i,k)
        do i=1,n
           a(i+i*(i-1)/2)= b(i)+1
           do k=1,i
              c(i+i*(i-1)/2+k)=d(i,k)+1
           end do
        end do
        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
do i=1,n
   b(i)=(a(j)+a(i))/2.
   j=i
end do
```

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) then
         b(1)=(a(j)+a(1))/2.
c$omp    parallel do private (i),lastprivate(j)
         do i=2,n
            j=i-1
            b(i)=(a(j)+a(i))/2.
         end do
      end if
```

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
      do i=1,n
         b(i)=(a(j)+a(i))/2.
         j=j+1
      end do
```

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

```
      j=n
      do i=1,n
         b(i)=(a(m)+a(i))/2.
         m=i+j
      end do
      j=n+j
```

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) then
         b(1)=(a(j)+a(i))/2.
c$omp    parallel do private (i,m)
         do i=2,n
            m=i-1+j
            b(i)=(a(m)+a(i))/2.
         end do
         j=n+j   !this has to be inside the if
      end if
```

4.5 Ordered sections.

Sometimes the only way to avoid races is to execute the code serially. Consider the looP:

```
do i=1,n
   a(i)=b(i)+1
   c(i)=sin(c(i-1))+a(i)
   d(i)=c(i)+d(i-1)**2
end do
```

Although there is no clear way to avoid races in this loop, we could execute in parallel the first statement. In fact, we can in this case transform the loop into:

```
c$omp   parallel do
        do i=1,n
           a(i)=b(i)+1
        end do
        do i=1,n
           c(i)=sin(c(i-1))+a(i)
           d(i)=c(i)+d(i-1)**2
        end do
```

However, there is a way to improve the performance of the whole loop with the `ordered` directive whose syntax is as follows:

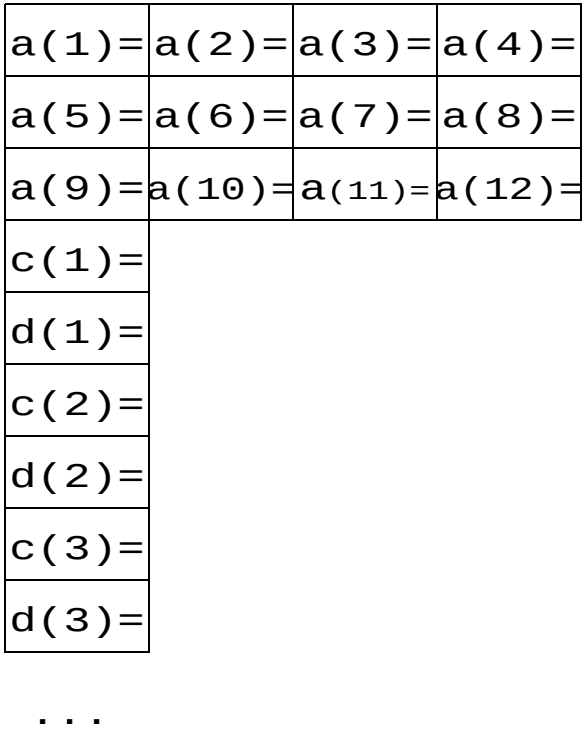
```
c$omp   ordered[(name)]
           block
c$omp   end ordered[(name)]
```

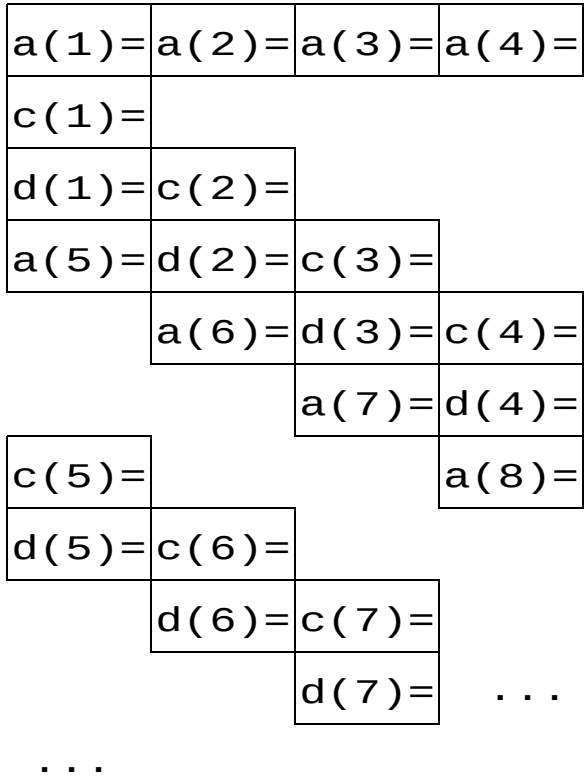
The interleaving of the statements in the ordered sections of different iterations are identical to that of the sequential program. Ordered sections without a name are all assumed to have the same name.

Thus, the previous loop can be rewritten as:

```
c$omp   parallel do
           do i=1,n
              a(i)=b(i)+1
c$omp         ordered (x)
                 c(i)=sin(c(i-1))+1
c$omp         end ordered(x)
c$omp         ordered (y)
                 d(i)=c(i)+d(i-1)**2
c$omp         end ordered (y)
      end do
```

Thus, we have two ways of executing the loop in parallel. Assuming n=12, and four processors, the following time lines are feasible:





Notice that now no races exist because accesses to the same memory location are always performed in the same order.

Ordered sections may need to include more than one statement. For example, in the loop:

```
do i=1,n
   ...
   a(i)=b(i-1)+1
   b(i)=a(i)+c(i)
   ...
end do
```

the possibility of races would not be avoided unless both statements are made part of the same ordered section.

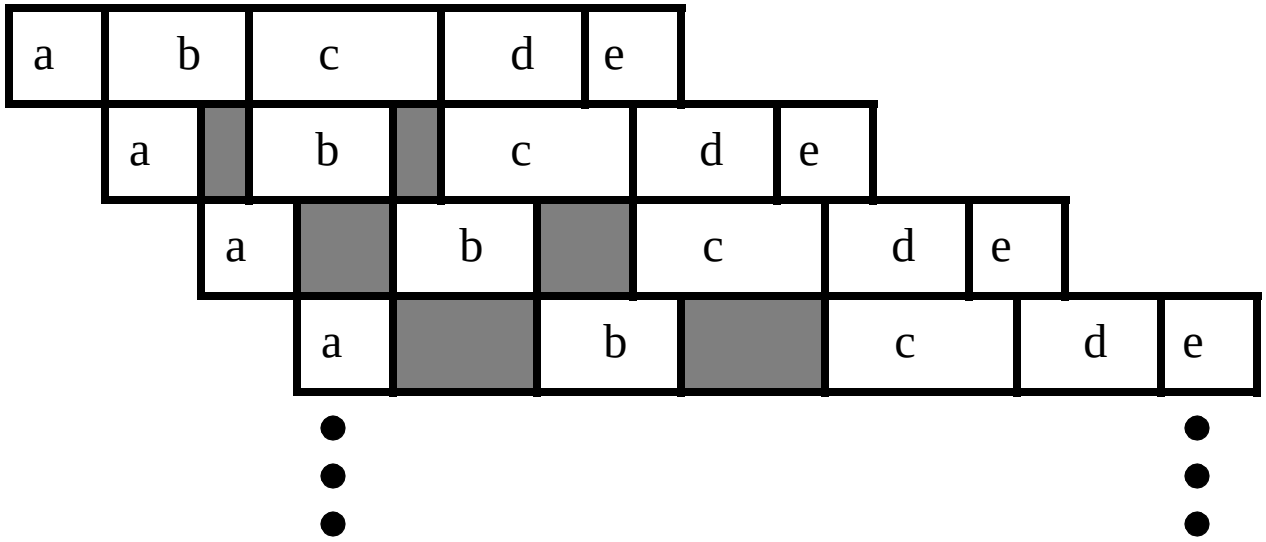
It is important to make the ordered sections as small as possible because the overall execution time depends on the size of the longest ordered section.

4.6 Execution time of a `parallel do` when `ordered` sections have constant execution time.

- Consider the loop

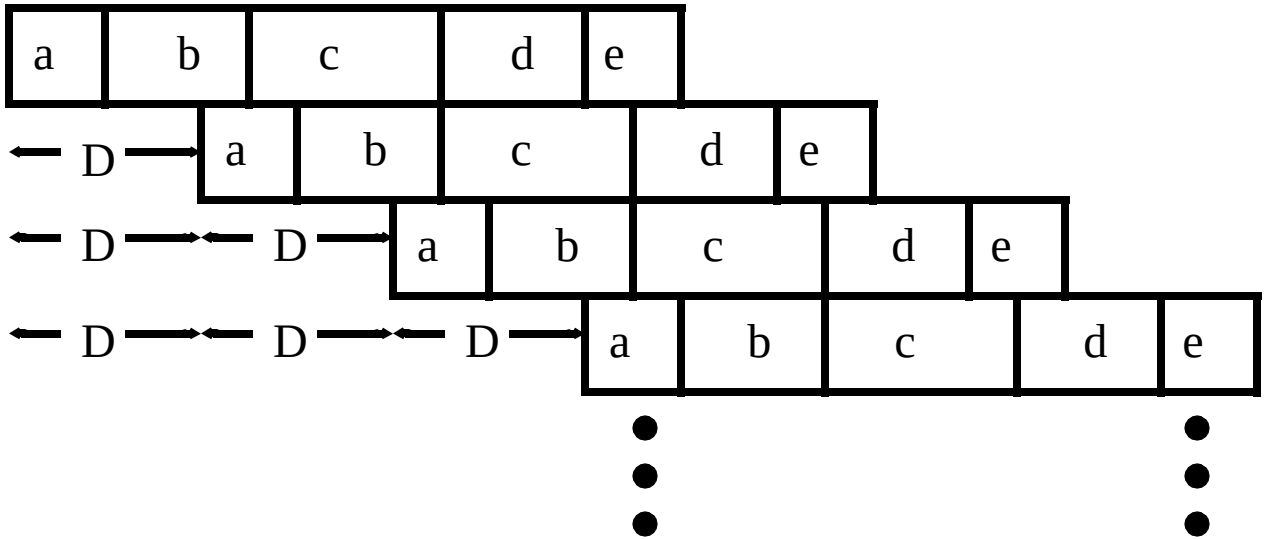
```
c$omp   parallel do
        do i=1,n
c$omp      ordered (a)
              aa = ...
c$omp      end  ordered (a)
c$omp      ordered (b)
              ...
c$omp      ordered (c)
              ...
c$omp      ordered (d)
              ...
c$omp      ordered (e)
              ...
   end do
```

- Assume its execution time lines have the following form:



which, in terms of performance, is equivalent to the following

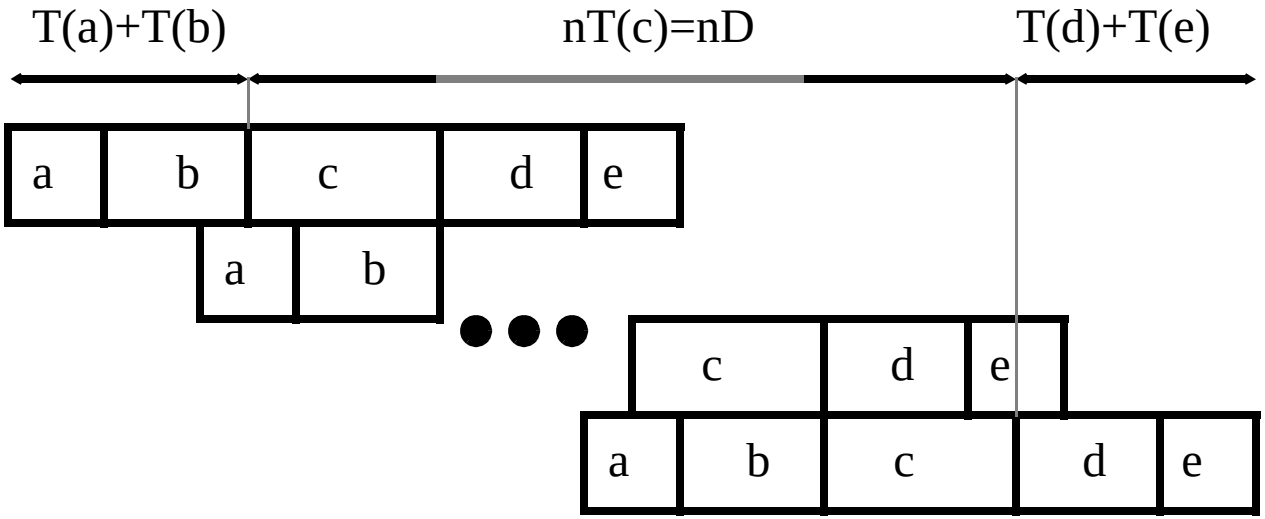
time lines:



where a constant delay D between the start of consecutive iterations is evident. This delay is equal to the time of the longest ordered section (i.e., D=T(c) in this case).

- The execution time of the previous loop using n processors is: T(a)+T(b)+nT(c)+T(d)+T(e)

  as can be seen next:



- In general the execution time when there are as many processors as iterations is

  $$nD+(B-D)=(n-1)D+B$$

  where B is the execution time of the whole loop body.

4.7 Critical Regions and Reductions

Consider the following loop:

```
do i=1,n
   do i=1,m
      ia(i,j)=b(i,j)+d(i,j)
      isum=isum+ia(i,j)
   end
end
```

Here, we have a race due to `isum`. This race cannot be removed by the techniques discussed above. However, the `+` operation used to compute `isum` is associative and `isum` 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:

```
c$omp   parallel private(local_isum)
        local_isum=0
c$omp   pdo
        do i=1,n
           do j=1,m
              local_isum=local_isum + ia(j,i)
           end do
        end do
c$omp   end pdo nowait
c$omp   critical
        isum=isum+local_isum
c$omp   end critical
c$omp   end parallel
```

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

The statement

```
isum=isum+local_isum
```

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

```
load    register_1,isum
load    register_2,local_isum
add     register_3,register_1,register_2
store   register_3,isum
```

Assume now there are two tasks executing the statement

```
isum=isum+local_isum
```

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

time	task 1	isum	task 2
1	load r1,local_isum	0	
2	load r2, isum	0	load r1,local_isum
3	add r3,r2,r1	0	load r2,isum
4	store r3, isum	10	add r3,r2,r1
		15	store r3,isum

As can be seen, interleaving the instructions between the two 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.

The assumption is that it does not matter in which order the tasks enter a critical region as long as they are never inside a critical region