5.5 Ordered sections.

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

```fortran
do i=1,n
   a(i)=b(i)+1
   c(i)=sin(c(i-1))+1
   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:

```fortran
!omp parallel do
   do i=1,n
      a(i)=b(i)+1
   end do
   do i=1,n
      c(i)=sin(c(i-1))+1
      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
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
  do  i=1,n
     a(i)=b(i)+1
  omp ordered (x)
     c(i)=sin(c(i-1))+1
  omp end ordered(x)
  omp ordered (y)
     d(i)=c(i)+d(i-1)**2
  omp end ordered (y)
  end do
```
Thus, we have two ways of executing the loop in parallel. Assuming $n=12$ and four processors, we would have the following time lines in each case:

\[
\begin{array}{cccc}
\text{a}() &=& \text{a}() &=& \text{a}() \\
\text{a}() &=& \text{a}() &=& \text{a}() \\
\text{a}() &=& \text{a}() &=& \text{a}() \\
\text{c}() &=& \text{c}() &=& \text{c}() \\
\text{d}() &=& \text{d}() &=& \text{d}() \\
\text{c}() &=& \text{c}() &=& \text{c}() \\
\end{array}
\]
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:

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

- Consider the loop

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

\[ T(a) + T(b) \quad nT(c) = nD \quad T(d) + T(e) \]

- 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.
5.6 Critical Regions and Reductions

Consider the following loop:

\[
\begin{align*}
&\text{do } i=1,n \\
&\quad \text{do } i=1,m \\
&\quad \quad \text{ia}(i,j) = b(i,j) + d(i,j) \\
&\quad \quad \text{isum} = \text{isum} + \text{ia}(i,j) \\
&\quad \text{end} \\
&\text{end}
\end{align*}
\]

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

```plaintext
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
\[ \text{isum} = \text{isum} + \text{local}_\text{isum} \]

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

- load \( \text{register}_1, \text{isum} \)
- load \( \text{register}_2, \text{local}_\text{isum} \)
- add \( \text{register}_3, \text{register}_1, \text{register}_2 \)
- store \( \text{register}_3, \text{isum} \)

Assume now there are two tasks executing the statement
\[ \text{isum} = \text{isum} + \text{local}_\text{isum} \]

simultaneously. In one \( \text{local}_\text{sum} \) is 10, and in the other 15. Assume \( \text{isum} \) 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 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 of the same name at the same time.
An alternative way of writing the above parallel loop is:

```c
omp parallel do reduction(+:isum)
do i=1,n
do j=1,n
    isum=isum+ia(j,i)
end do
end do
```

The reduction clause can be applied to a number of operations and intrinsic functions.
Chapter 6. Parallel Vector Algorithms
6.1 Introduction

We will now study several algorithms where the parallelism can be easily expressed in terms of vector operations.

Simplistic timing figures will be given in some cases for pipelined machines, array machines, and multiprocessors.

In these timings, parallelism overhead, subscript computations, and memory access/communications costs will be ignored.
6.2 Time to execute a vector operation

Let us start with the simplest possible situation. Consider the following generic vector operation:

\[ a(1:n) \# b(1:n) \]

First, let us assume a pipelined arithmetic unit with \( s \) stages for operation \( \# \). Each stage takes \( \tau \) units of time.

The time to execute the vector operation under these assumptions is:

\[ t_{\text{pipeline}} = (s + (n - 1))\tau \]

Compare this with the serial time when no pipelining takes place:

\[ t_{\text{serial}} = s\tau n \]
Consider now an array machine with $P$ arithmetic units, or a multiprocessor with $P$ processors.

The execution time is:

$$t_{parallel} = \left\lfloor \frac{n}{P} \right\rfloor t_#$$

where $t_#$ is the time to execute one # operation.

In a system where each processor contains an arithmetic pipeline, the execution time would be:

$$t_{parallel\&pipeline} = \left( (s - 1) + \left\lfloor \frac{n}{P} \right\rfloor \right) \tau$$
6.3 Time to Execute a Reduction

\[ s = a(1) + a(2) + a(3) + \ldots a(n) \]

A sequence of \( \lceil \log_2 n \rceil \) vector operations of length \( n/2, n/4, \ldots, 1 \) suffices to compute the reduction (assuming associativity).

Therefore (assuming \( n=2^m \)):

\[
t_{pipeline} = \sum_{i=1}^{\log n} \left( s - 1 + \left( \frac{n}{2^i} \right) \right) \tau = ((s-1)\log n + (n-1))\tau
\]
In the case of an array machine or multiprocessor, there are two cases. First, if \( P < n \), and if we follow the approach presented in our discussion of reductions in OpenMP, we have:

\[
    t_{parallel} = \left\lfloor \frac{n}{P} \right\rfloor t_+ + (P - 1)t_+
\]

If the final reduction is done in logarithmic time, the execution time is:

\[
    t_{parallel} = \left\lfloor \frac{n}{P} \right\rfloor t_+ + \left\lceil \log P \right\rceil t_+
\]

If \( P \geq n \), the time is:

\[
    t_{parallel} = \left\lceil \log n \right\rceil t_+
\]
6.4 Parallel Prefix

Consider the following loop:

\[
A(0) = 0 \\
\text{DO } I = 1, N \\
\quad A(I) = A(I-1) + B(I) \\
\text{END DO}
\]

The loop seems sequential because each iteration needs information on the value computed in the preceding iteration.

However, we can use a parallel prefix approach to compute the value of vector A in parallel as follows:

\[
B(1) \\
\downarrow \\
B(1) \\
\downarrow \\
B(1) + B(2) \\
\downarrow \\
B(1) + B(2) + B(3) \\
\downarrow \\
B(1) + B(2) + B(3) + B(4) \\
\downarrow \\
\ldots \\
B(N-1) + B(N)
\]

\[
B(1) \\
\downarrow \\
B(1) + B(2) \\
\downarrow \\
B(1) + B(2) + B(3) \\
\downarrow \\
B(1) + B(2) + B(3) + B(4) \\
\downarrow \\
\ldots \\
B(N-3) + B(N-2) + B(N-1) + B(N)
\]
A parallel program implementing this strategy under the assumption that $N=2^k$ is:

\[ A(1:N) = B(1:N) \]
\[ \text{DO } I = 1, K-1 \]
\[ \quad A(2^I:N) = A(2^I:N) + A(1:N-2^I+1) \]
\[ \text{END DO} \]

For an array machine with the number of processing units $P \geq n$:

\[ t_{\text{parallel}} = t + \lfloor \log n \rfloor \]
6.4 Relative Performance

How much faster does a program run when executed in parallel?

Speedup: \( S_P = \frac{T_1'}{T_P} \)  \hspace{1cm} (1)

\( T_1' \): Execution time of the program on a single (scalar) processor.

\( T_P \): Execution time on a parallel machine.

Parallel programs may introduce some redundancy to achieve higher parallelism. In a sequential program, the goal is to minimize the total number of operations because this number is directly related to the execution time. In a parallel program, this relationship is not direct. For this reason a more honest formula for speedup is:

Speedup: \( S_P = \frac{T_1}{T_P} \)  \hspace{1cm} (2)

where \( T_1 \) is the best known serial version of the program.

The speedup in (1) is known as the parallel speedup.
Assume a multiprocessor with P processors or an array machine with P processing elements. The speedup can be linear in P (that is, of the form k*P for k <= 1), logarithmic (that is, of the form k * log P), or it can have many other forms. In a real machine the speedup is seldom a nice function of the number of processors.

In some cases the speedup is superlinear; that is, the speedup is greater than p for p processors. This happens when, for example, each processor has its own cache memory. In this way using several processors also increases the size of the cache memory. Another case when you can get superlinear speedup is in program performing some form of search operation.

Other important measures include:

1. Efficiency: \[ E_P = \frac{T_1}{PT_P} \]

where P is the number of processors if the target machine is a multiprocessor (assuming single-user mode) or the number of processing elements in an array processor.
2. Redundancy: \[ R_P = \frac{O_P}{O_1} \]

where \( O_P \) is the number of operations in the parallel program, and \( O_1 \) is the number of operations in the best known serial version.
6.5 Examples of Speedup and Efficiency

Consider

\[ a(1:n) + b(1:n) \]

The speedup, efficiency, and redundancy on a pipelined unit are:

\[
S_s = \frac{s\tau n}{\tau[s + (n-1)]} = \frac{sn}{s + n - 1} \rightarrow s
\]

\[
E_s = \frac{s\tau n}{s\tau[s + (n-1)]} = \frac{n}{s + (n-1)} < 1
\]

\[
R_s = \frac{O_s}{O_1} = \frac{ns}{ns} = 1
\]
In an array machine or multiprocessor:

\[ t_{\text{parallel}} = \left\lceil \frac{n}{P} \right\rceil t_+ \]

\[ S_P = \frac{nt_+}{\left\lceil \frac{n}{P} \right\rceil t_+} = \frac{n}{\left\lceil \frac{n}{P} \right\rceil} \]

The value of \( S_P \) is \( P \) if \( n \) is a multiple of \( P \).

\[ E_P = \frac{nt_+}{P \left\lceil \frac{n}{P} \right\rceil t_+} = \frac{n}{P \left\lceil \frac{n}{P} \right\rceil} \]

\( E_P \) is 1 if \( n \) is a multiple of \( P \). Otherwise it is \( < 1 \).

\[ R_P = 1 \]
The speedup, efficiency, and redundancy of the parallel prefix example on an array machine or multiprocessor with \( P=n \) are:

\[
S_n = \frac{nt_+}{\lceil \log n \rceil t_+} = \frac{n}{\lceil \log n \rceil}
\]

\[
E_n = \frac{nt_+}{n\lceil \log n \rceil t_+} = \frac{1}{\lceil \log n \rceil}
\]

\[
R_n = \frac{O_n}{O_1} = \frac{(n-1) + (n-2) + \ldots + \left(\frac{n}{2}\right)}{n} = \frac{n(\log n - 1) + 1}{n-1} \approx \log n
\]
6.6 Amdahl’s Law

Assume a program which executes in one of two modes: serial or perfectly parallel. In the perfectly parallel mode, as many processors as desired can cooperate in the execution of the program. Assume that $s$ is the fraction of the program that is serial and $q$ is the fraction that is parallel. The speedup of this program, given $p$ processors is then

$$S_p = T \left( s + q \right) / T(s + q / P) = 1 / (s + q /P)$$

When $P$ is large, the speedup curve is very steep near $s = 0$. To obtain a very high speedup, the serial fraction of the program has to be very small.

The form of this curve has been used to argue that is difficult to obtain good speedups. However, there are many examples where good speedups are obtained (see Gustafson: Reevaluating Amdahl’s law. CACM Vol 31, No. 5. pp. 532-533). The argument is that the problem size tends to grow with the number of processors. If this is the case, we have:

$$\text{Scaled speedup} = (s + q * P)/(s+(q*P)/P) = P + (1-P) * s$$

This is a line with a moderate slope.
6.6 Matrix-Vector Multiplication

In mathematical notation:

\[
\begin{bmatrix}
  A_{11} & A_{12} & \ldots & A_{1n} \\
  A_{21} & A_{22} & \ldots & A_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  A_{m1} & A_{m2} & \ldots & A_{mn}
\end{bmatrix}
\begin{bmatrix}
  V_1 \\
  V_2 \\
  \vdots \\
  V_n
\end{bmatrix}
= \begin{bmatrix}
  \sum_{i=1}^{n} A_{1i}V_i \\
  \sum_{i=1}^{n} A_{2i}V_i \\
  \vdots \\
  \sum_{i=1}^{n} A_{mi}V_i
\end{bmatrix}
\]

In Fortran:

do i=1,m
  R(i) = 0
  do j=1,n
    R(i) = R(i) + A(i,j) * V(j)
  end do
end do
\begin{verbatim}
do i=1,m
    R(i) = DOT_PRODUCT(A(i,1:n), V(1:n))
end do

The dot product is a vector multiplication (of length \( n \), in this case) followed by a reduction.

Time in a pipelined machine for a dot product:

\[(s + n - 1)\tau + ((s - 1)\log n + (n - 1))\tau = [(s - 1)\log n + s + 2(n - 1)]\tau\]

The total time for the matrix-vector multiplication is then:

\[m[(s - 1)\log n + s + 2(n - 1)]\tau\]

In an array machine or in a multiprocessor, the time if \( P>n \) is:

\[(t + m[\lceil \log n \rceil + 1])\]
\end{verbatim}
Alternatively, by interchanging the loop headers, the program could be written as follows:

\[
\begin{align*}
\text{do } & j=1,n \\
& \quad \text{do } i=1,m \\
& \quad \quad R(i) = R(i) + A(i,j) \times V(j) \\
& \quad \text{end do} \\
& \text{end do}
\end{align*}
\]

This leads to the following sequence of vector operations:

\[
\begin{align*}
\text{do } & j=1,n \\
& \quad R(1:m) = R(1:m) + A(1:m,j) \times V(j) \\
& \text{end do}
\end{align*}
\]

The time for this loop in a pipelined machine is:

\[n(s + m - 1)\tau\]

In an array machine or in a multiprocessor, the time (if \(P > m\)) is:

\[t + 2n\]
6.7 Matrix Multiplication

1. *Inner product method.*

Matrix multiplication is usually written:

```fortran
do i=1,n
  do j=1,n
    do k=1,n
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    end do
  end do
end do
```

The most direct translation of this program into vector form is:

```fortran
do i=1,n
  do j=1,n
    C(i,j)=DOT_PRODUCT(A(i,1:n),B(1:n,j))
  end do
end do
```
The time on a pipelined machine is:

\[ n^2((s - 1)\log n + s + 2(n - 1))\tau \]

The time on an array machine or multiprocessor if \( P > n \) is:

\[ (t_+[\log n] + t_*)n^2 \]
2. *Middle-product method* (n-parallelism)

This is obtained by interchanging the headers in the original matrix multiplication loop.

\[
\begin{align*}
&\text{do } j=1,n \\
&\quad \text{do } k=1,n \\
&\quad \quad \text{do } i=1,n \\
&\quad \quad \quad C(i,j)=C(i,j)+A(i,k) \cdot B(k,j) \\
&\quad \quad \text{end do} \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]

The direct translation of this loop into vector form is:

\[
\begin{align*}
&\text{do } j=1,n \\
&\quad \text{do } k=1,n \\
&\quad \quad C(1:n,j)=C(1:n,j)+A(1:n,k) \cdot B(k,j) \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]
Alternatively, the headers could have been exchanged in a different order to obtain the loop:

\[
\begin{align*}
\text{do } & j=1,n \\
& \quad \text{do } k=1,n \\
& \quad \quad C(i,1:n) = C(i,1:n) + A(i,k) \times B(k,1:n) \\
& \quad \text{end do}
\end{align*}
\]

end do

The time on a pipelined machine is:

\[
2n^2(s + (n - 1))\tau
\]

The time in an array machine is:

\[
(t_+ + t_*)n^2
\]