Parallel Vector Algorithms
Introduction

Next, we study several algorithms where parallelism can be easily expressed in terms of array operations. We will use Fortran 90 to represent these algorithms.

Simplistic timing figures will be given in some cases for array machines.

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

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

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

Consider also an array machine with \( P \) arithmetic units. The execution time is:

\[ t_{\text{parallel}} = \left\lfloor \frac{n}{P} \right\rfloor t_{\#} \]

where \( t_{\#} \) is the time to execute one \( \# \) operation.
Reductions in Fortran 90

A typical reduction is \texttt{sum(array)} which returns, as we should expect, the sum of the elements of an integer, real, or complex array. It returns zero if \texttt{array} has size zero.

Others include:

\begin{itemize}
  \item \texttt{all(mask)} Returns the logical value \texttt{.true.} if all elements of the logical array \texttt{mask} are \texttt{.true.} or \texttt{mask} has size zero, and otherwise returns the value \texttt{.false.}.
  \item \texttt{any(mask)} Returns the logical value \texttt{.true.} if any of the elements of the logical array \texttt{mask} is \texttt{.true.}, and returns the value \texttt{.false.} if no elements are \texttt{.true.} or if \texttt{mask} has size zero.
  \item \texttt{count(mask)} Returns the number of \texttt{.true.} values in \texttt{mask}.
  \item \texttt{maxval(array)} Returns the maximum value of the elements of an integer or real array.
\end{itemize}
minval(array)  Returns the minimum value of the elements of an integer or real array.

product(array) returns the product of the elements of an integer, real, or complex array. It returns 1 if array has size zero.

All these functions have an optional argument dim if this is present, the operation is applied to all rank-one sections that span right through dimension dim to produce an array of rank reduced by one and extends equal to the extents in the other dimensions. For example, if a is a real array of shape [4,5,6], sum(a, dim=2) is a real array of shape [4,6] and element (i,j) has value sum(a(i,:,j)).

The functions maxval, minval, product, and sum have a third optional argument, mask. If this is present, it must have the same shape as the first argument and the operation is applied to the elements corresponding to true elements of mask; for example, sum(a, mask=a>0) sums the positive elements of the array a.
Two other useful Fortran 90 functions.

1. `spread(source, dim, ncopies)`

Returns an array of the same type as `source` but with rank increased by one over `source`. `Source` may be a scalar or an array. `Dim` and `ncopies` are integer scalars. The result contains \( \max(ncopies, 0) \) copies of `source`, and element \((r_1,\ldots,r_{n+1})\) of the result is `source` \((s_1,\ldots,s_n)\) where \((s_1,\ldots,s_n)\) is \((r_1,\ldots,r_{n+1})\) with subscript `dim` omitted (or `source` itself if it is a scalar).

Example of use:

\[
\begin{align*}
    a &= \text{spread}(x, \text{dim}=2, \text{ncopies}=n) + \text{spread}(x, 1, n) \\
    w &= \text{sum(abs}(a), \text{dim}=1)
\end{align*}
\]
is equivalent to:

\[
\begin{align*}
\text{do } & i=1,n \\
&w(i)=0 \\
\text{do } & j=1,n \\
&w(i)=w(i)+\text{abs}(x(i)+x(j)) \\
\text{end } & \text{do} \\
\text{end } & \text{do}
\end{align*}
\]

2. `maxloc(array[,mask])`

Returns a rank-one integer array of size equal to the rank of `array`. Its value is the subscript of an element of maximum value.
Time to Execute a Reduction

Consider a reduction such as:
\[ r = \text{sum}(a(1:n)) = a(1) + a(2) + a(3) + \ldots + a(n) \]

or, in general
\[ r = a(1) \# a(2) \# a(3) \# \ldots \# a(n) \]

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 of the \# operation).

In the case of an array machine, there are two cases. First, if \( P < n/2 \), and if we follow the approach presented in our discussion of reductions in OpenMP, we have:

\[
 t_{\text{parallel}} = \left( \left\lceil \frac{n}{P} \right\rceil - 1 \right) t_+ + (P - 1) t_+
\]
If the final reduction can also be done in logarithmic time using a reduction tree approach:

\[ t_{parallel} = \left( \left\lceil \frac{n}{P} \right\rceil - 1 \right) t_+ + \lceil \log P \rceil t_+ \]

In this case, the execution time is:

If \( P \geq n/2 \), the time is:

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

Consider the following loop:

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

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:
A parallel program implementing this strategy under the assumption that $N=2^k$ is:

\[
\begin{align*}
A(1:N) &= B(1:N) \\
\text{DO } I = 0, K-1 \\
& \quad A(2^{*I}+1:N) = A(2^{*I}+1:N) + A(1:N-2^{*I}+1) \\
\text{END DO}
\end{align*}
\]

For an array machine with the number of processing units $P \geq n-1$:

\[
t_{\text{parallel}} = t_{+}[\log n]
\]

As in the case of reduction, parallel prefix can be applied to any associative binary operation.
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 \textit{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 \( k \times P \) for \( k \leq 1 \)), logarithmic (that is, of the form \( k \times \log P \)), 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.
Examples of Speedup and Efficiency

Consider

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

The speedup, efficiency, in an array machine:

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

\[ S_P = \frac{nt_+}{\frac{n}{P} t_+} = \frac{n}{\frac{n}{P}} \]

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

\[ E_P = \frac{nt_+}{\frac{Pn}{P} t_+} = \frac{n}{\frac{Pn}{P}} \]
$E_P$ is 1 if $n$ is a multiple of $P$. Otherwise it is < 1. The speedup,

$$R_P = 1$$

efficiency, and redundancy of the parallel prefix example on an array machine with $P=n$ are:

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

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

$$R_n = \frac{O_n}{O_1} = \frac{(n-1) + (n-2) + \ldots + \left(\frac{n-1}{2}\right)}{n} = \frac{n(\log n - 1) + 1}{n-1} \approx \log n$$
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
```
The inner loop performs a dot product (or inner product) of two vectors. It can be represented in Fortran 90 as follows:

```fortran
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.

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

\[
(m(\lceil \log n \rceil t_* + t_*))
\]

Alternatively, by interchanging the loop headers, the program could be written as follows:

```fortran
do j=1,n
    do i=1,m
        R(i) = R(i) + A(i,j) * V(j)
    end do
end do
```
This leads to the following sequence of vector operations:

\[
\begin{align*}
&\text{do } j=1, n \\
&\quad R(1:m) = R(1:m) + A(1:m, j) \times V(j) \\
&\text{end do}
\end{align*}
\]
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 an array machine or multiprocessor if $P > n$ is:

$$
(t_+\lceil\log n\rceil + t_*)n^2
$$
2. *Middle-product method* (n-parallelism)

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

```plaintext
do  j=1,n
   do  k=1,n
      do  i=1,n
         \[ C(i, j) = C(i, j) + A(i, k) \times B(k, j) \]
      end do
   end do
end do
```

The direct translation of this loop into vector form is:

```plaintext
do  j=1,n
   do  k=1,n
      \[ C(1:n, j) = C(1:n, j) + A(1:n, k) \times B(k, j) \]
   end do
end do
```
Alternatively, the headers could have been exchanged in a different order to obtain the loop:

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

The time in an array machine is:

\[ (t_+ + t_\ast) n^2 \]
3. Outer-product method (n²-parallelism)

Another interchange of the loop headers produce:

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

To obtain n² parallelism, the inner two loops should take the form of a matrix operation:

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

Where the operator \( \otimes \) represents the outer product of two vectors. Given two vectors \( a \) and \( b \), their outer product is a matrix \( Z \) such that \( Z_{i,j} = a_i \times b_j \). Notice that the previous loop is NOT a valid Fortran or Fortran 90 loop because \( \otimes \) is not a valid Fortran character.
The outer product matrix in the loop above has the following form:

\[
\begin{bmatrix}
A_{1k}B_{k1} & A_{1k}B_{k2} & A_{1k}B_{k3} & \ldots \\
A_{2k}B_{k1} & A_{2k}B_{k2} & A_{2k}B_{k3} & \ldots \\
A_{3k}B_{k1} & A_{3k}B_{k2} & A_{3k}B_{k3} & \ldots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}
\]

This matrix is the element-by-element product of the following two matrices:

\[
\begin{bmatrix}
A^k & A^k & \ldots & A^k
\end{bmatrix}
\begin{bmatrix}
B^k \\
B^k \\
\vdots \\
B^k
\end{bmatrix}
\]

which are formed by replicating \( A^k = A(1:n, k) \) and \( B^k = B(k, 1:n) \) along the appropriate dimensions. This replication can be achieved using the Fortran 90 \texttt{SPREAD} function discussed above:
spread(A(1:n,k),dim=2,ncopies=n)*spread(B(k,1:n),dim=1,ncopies=n))

The resulting loop is therefore:

\[
do \ k=1, n \\
C=C+SPREAD(A(1:n,k),2,n)*SPREAD(B(k,1:N,1,n) \\
end do
\]

In an array machine with P>n^2, the time would be:

\[
(2t_{copy}[log n]+t_*+t_+[log n])n
\]

where \( t_{copy} \) is the time to copy a vector. The time to spread to \( n \) copies is logarithmic as discussed in class.
4. $n^3$ parallelism

The product of two $n \times n$ matrices, $C = \text{matmul}(A, B)$, can be computed by adding $n$ matrices of rank $(n, n)$:
These $n$ matrices of rank $(n,n)$ can be computed by multiplying (element-by-element) two three-dimensional arrays of rank $(n,n,n)$.

The two three-dimensional arrays are formed by replicating $A$ and $B$ along different dimensions as shown next:

![Diagram of array replication]

This replication can, again, be achieved, with $\text{SPREAD}$. 
Thus, give the following three directions of replication:

we can start by computing a \( n^3 \) temporary array \( T \) as follows:

\[
T(:, :, :) = \text{SPREAD}(A, \text{DIM}=3, \text{NCOPIES}=n) \times \text{SPREAD}(B, \text{DIM}=1, \text{NCOPIES}=n)
\]

Then, the result is just \( C = \text{SUM}(T, \text{DIM}=2) \)

In an array machine with \( P \geq n^3 \) processing unit, the time to compute \( C \) would be:

\[
(2t_{\text{copy}} \lceil \log n \rceil + t_\star + t_+)\]
Multiplication by Diagonals

An \( n \times n \) matrix \( A \) is banded if \( A_{ij} = 0 \) for \( i-j \geq \beta_1, j-i \geq \beta_2 \):

\[
\begin{bmatrix}
A_{11} & A_{12} & \cdots & A_{1, \beta_2} & 0 & 0 & 0 \\
\vdots & A_{22} & A_{23} & \cdots & A_{2, \beta_2 + 1} & 0 & 0 \\
A_{\beta_1, 1} & \cdots & \cdots & \cdots & \cdots & \cdots & A_{n-\beta_2 + 1, n} \\
0 & A_{\beta_1 + 1, 2} & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & \cdots & \cdots & \cdots & \cdots & A_{n-1, n} \\
0 & 0 & 0 & A_{n,n-\beta_1 + 1} & \cdots & \cdots & A_{n,n}
\end{bmatrix}
\]

For a small band, for example, \( \beta_1 = \beta_2 = 3 \), the algorithm discussed before for matrix-vector multiplication is not efficient.
An alternative is to do the product by diagonals:

$$\begin{bmatrix}
A_0 & A_1 & \cdots & A_{\beta_2} & 0 & 0 & 0 \\
A_{-1} & & & & 0 & 0 & 0 \\
& \ddots & & & \ddots & 0 & 0 \\
& & \ddots & & \ddots & \ddots & 0 \\
& & & \ddots & \ddots & \ddots & \ddots \\
& & & & \ddots & \ddots & \ddots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots 
\end{bmatrix} \times V$$

After separating the diagonals into separate matrices, we get:

$$\begin{bmatrix}
A_0 & 0 & 0 & 0 & 0 \\
0 & A_1 & 0 & 0 & 0 \\
0 & 0 & \ddots & \ddots & \ddots \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \cdots 
\end{bmatrix} V + \begin{bmatrix}
0 & A_1 & 0 & 0 & 0 \\
0 & 0 & \ddots & \ddots & \ddots \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 
\end{bmatrix} V + \cdots + \begin{bmatrix}
0 & 0 & 0 & A_{\beta_2} & 0 \\
0 & 0 & 0 & 0 & \ddots \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 
\end{bmatrix} V + \cdots + \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
A_{\beta_2} & 0 & 0 & 0 & 0 
\end{bmatrix} V + \cdots + \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 
\end{bmatrix} V$$
which can be written as follows:

\[ A_0 V^\wedge A_1 V^2 + \ldots + A_{\beta_2} V^{\beta_2} + A_{-1} V_{n-1} \vee \ldots \vee A_{-\beta_1} V_{n-\beta_1} \]

where \( V^j = (V_j, \ldots, V_n) \) and \( V_{n-j} = (V_1, \ldots, V_{n-j}) \).

Also, \( \wedge \) means add the sorter vector to the first component of the longer one, and \( \vee \) means add the shorter vector to the last component of the longer one.

In Fortran 90 (except for the greek letters and the subscripts):

\[
A_0 (1:n) * V(1:n) + \\
(\text{/ } A_1 (1:n-1) * V(2:n), 0. /) + \\
\ldots \\
(\text{/ } A_{\beta_1} (1:n-\beta_1) * V(\beta_1+1:n), (0., j=1, \beta_1) /) + \\
(\text{/ } 0., A_{-1} (1:n-1) * V(1:n-1) /) + \\
\ldots \\
(\text{/ } (0., j=1, \beta_2), A_{\beta_2} (1:n-\beta_2) * V(1:n-\beta_2) /)
\]