Chapter 8. Parallel Vector Algorithms

8.1 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 pipelined machines and array machines.

In these timings, subscript computations and memory access/ communications costs will be ignored.

8.2 Target machines

The natural target machines for programs with vector parallelism are array machines and pipelined processors.

It is also easy to exploit vector parallelism on shared memory multiprocessors and on message-passing multicomputers. However, the best parallel form for these machines sometimes cannot be expressed in vector notation.

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)
```

First, let us assume a pipelined arithmetic unit with $s_{\#}$ stages for operation #. Each stage takes τ units of time.

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

$$t_{pipeline} = (s_{\#} + (n - 1))\tau$$

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

$$t_{serial} = s_{\#}\tau n$$

Consider now an array machine with *P* arithmetic units.

The execution time is:

$$t_{parallel} = \left\lceil \frac{n}{P} \right\rceil t_{\#}$$

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

In a system where each processing unit contains an arithmentic pipeline, the execution time would be:

$$t_{parallel\ \&\ pipelined} = \left((s-1) + \left\lceil \frac{n}{P} \right\rceil \right) \tau$$

8.4 Reductions in Fortran 90

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

Others include:

`all(mask)`	Returns the logical value .true. if all elements of of the logical array `mask` are `.true.` or `mask` has size zero, and otherwise returns the value `.false.`
`any(mask)`	Returns the logical value `.true.` if any of the elements of the logical array `mask` is `.true.` , and returns the value `.false.` if no elements are `.true.` or if `mask` has size zero.
`count(mask)`	Returns the number of `.true.` values in `mask.`
`maxval(array)`	Returns the maximum value of the elements of an integer or real array.

`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 esample, `sum(a,mask=a>0)` sums the positive elements of the array a.

8.5 Two other useful Fortran 90 functions.

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

Returns an array of rhe 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:

```
a=spread(x,dim=2,ncopies=n)+spread(x,1,n)
w=sum(abs(a),dim=1)
```

is equivalent to:

```
      do i=1,n
         w(i)=0
         do j=1,n
            w(i)=w(i)+abs(x(i)+x(j))
         end do
      end do
```

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.

8.6 Time to Execute a Reduction

Consider a reduction such as:
`r = sum(a(1:n)) = a(1) + a(2) + a(3) + ... a(n)`

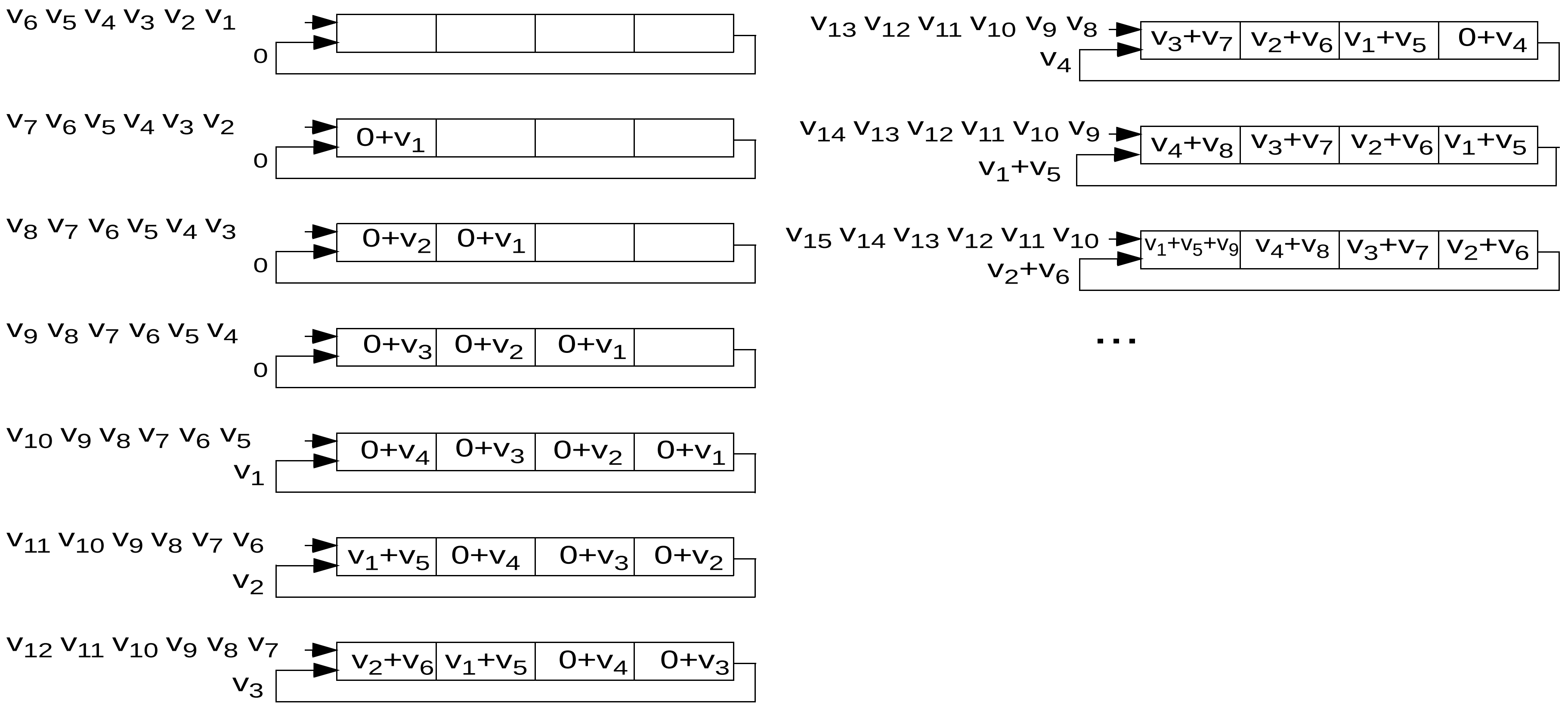
or, in general
`r =  a(1) # a(2) # a(3) # ... a(n)`

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

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

$$t_{pipeline} = \sum_{i=1}^{\log n} \left(s_{\#} - 1 + \left(\frac{n}{2^i} \right) \right) \tau = ((s_{\#} - 1) \log n + (n-1)) \tau$$

An alternative way of performing a reduction, which was implemented in the Cray-1, proceeded by feeding the pipeline elements of the vector to be reduced together with the output of the pipeline. Thus, in the case of `sum`, the elements of the input vector are added to the output of the pipeline as shown in the figure below. The pipeline is assumed to produce the identity of the reduction operation (zero in the case of a sum) until the first element of the vector exits the pipe.


$v_6\ v_5\ v_4\ v_3\ v_2\ v_1$
0
$v_7\ v_6\ v_5\ v_4\ v_3\ v_2$
0
$0+v_1$
$v_8\ v_7\ v_6\ v_5\ v_4\ v_3$
0
$0+v_2$ $0+v_1$
$v_9\ v_8\ v_7\ v_6\ v_5\ v_4$
0
$0+v_3$ $0+v_2$ $0+v_1$
$v_{10}\ v_9\ v_8\ v_7\ v_6\ v_5$
v_1
$0+v_4$ $0+v_3$ $0+v_2$ $0+v_1$
$v_{11}\ v_{10}\ v_9\ v_8\ v_7\ v_6$
v_2
v_1+v_5 $0+v_4$ $0+v_3$ $0+v_2$
$v_{12}\ v_{11}\ v_{10}\ v_9\ v_8\ v_7$
v_3
v_2+v_6 v_1+v_5 $0+v_4$ $0+v_3$
$v_{13}\ v_{12}\ v_{11}\ v_{10}\ v_9\ v_8$
v_4
v_3+v_7 v_2+v_6 v_1+v_5 $0+v_4$
$v_{14}\ v_{13}\ v_{12}\ v_{11}\ v_{10}\ v_9$
v_1+v_5
v_4+v_8 v_3+v_7 v_2+v_6 v_1+v_5
$v_{15}\ v_{14}\ v_{13}\ v_{12}\ v_{11}\ v_{10}$
v_2+v_6
$v_1+v_5+v_9$ v_4+v_8 v_3+v_7 v_2+v_6
...

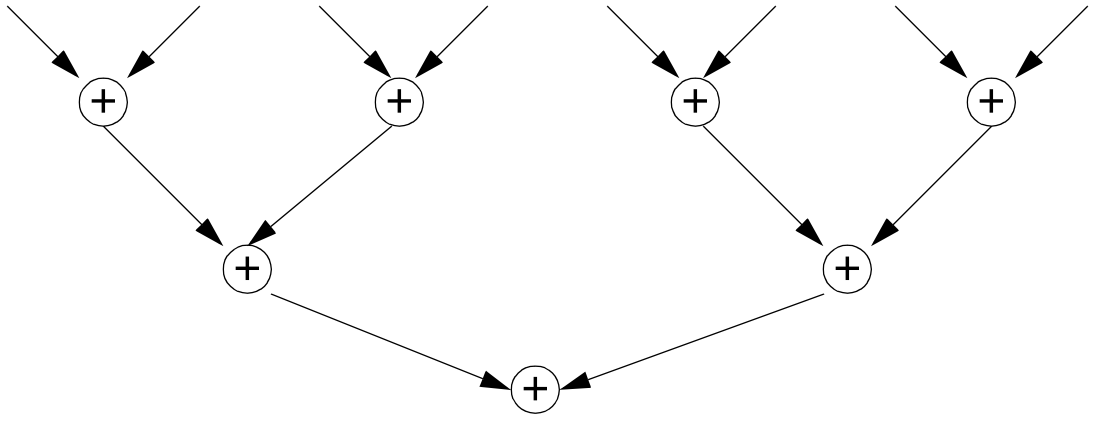
At the end of the process, there will be $s_{\#}$ (the number of stages used by the pipe to perform the # operation) partial results which should be added to get the final result. The time to get the final result is therefore:

$$t_{pipeline\ with\ feedback} = (s_{\#} + n - 1)\tau + (s_{\#} - 1)(s_{\#}\tau)$$

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:

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

If the final reduction can also be done in logarithmic time using a reduction tree approach:



In this case, the execution time is:

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

If *P >= n/2*, the time is:

$$t_{parallel} = \lceil \log n \rceil t_{+}$$

The # operation could be a simple arithmetic operation such as s + or * or it could be a more complex binary operation. For example, to implement `maxloc` in logarithmic time we could define an operation on two pairs consisting of a value and a location:

```
(v_1,loc_1) # (v_2,loc_2)=
        if v_1 < v_2 then return(v_1,loc_1)
                     else return(v_2,loc_2)
```

And, to implement an in logarithmic time an operation that finds the location of the first negative value in a vector we could define the following similar operation:

```
(v_1,loc_1) # (v_2,loc_2)=
        if v_1 < 0  then return(v_1,loc_1)
                    else return(v_2,loc_2)
```

Notice that both of these operations are associative (but NOT commutative).

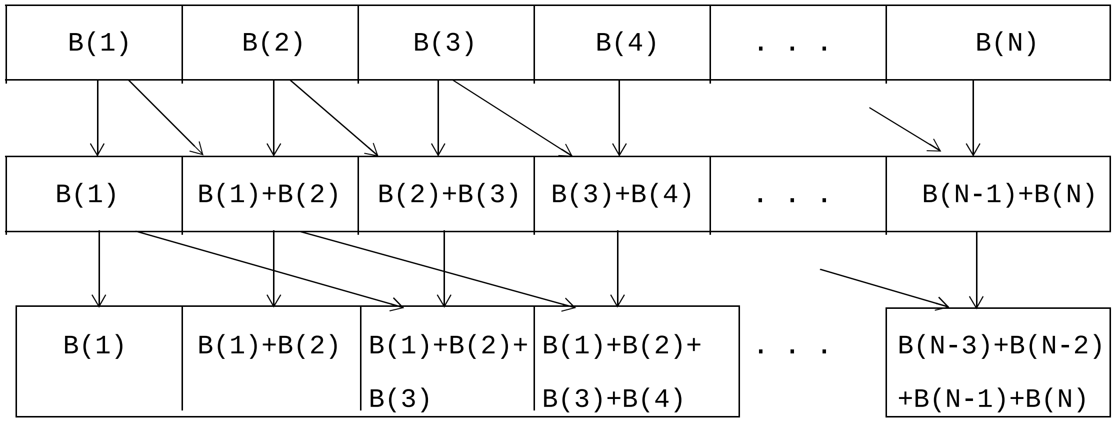
8.7 Parallel Prefix

Consider the following loop:

```
A(0)=0
DO I=1,N
   A(I)=A(I-1)+B(I)
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:



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

```
A(1:N)=B(1:N)
DO I=0,K-1
   A(2**I+1:N)=A(2**I+1:N)+A(1:N-2**I+1)
END DO
```

For an *array machine* with the number of procesing units *P>=n-1:*

$$t_{parallel} = t_+ \lceil \log n \rceil$$

As in the case of reduction, parallel prefix can be applied to any associative binary operation.

8.8 Relative Performance

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

Speedup: $S_P = T'_1 / T_P$ (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: $S_P = T_1 / T_P$ (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 superlinar speedup is in program performing some form of search operation.

Other important measures include:

1. Efficiency: $E_P = T_1 / PT_P$

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: $R_P = O_P/O_1$

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.

8.9 Examples of Speedup and Efficiency

Consider

```
a(1:n) + b(1:n)
```

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

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

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

$$R_s = \frac{O_s}{O_1} = \frac{ns}{ns} = 1$$

In an array machine:

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

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

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 with *P=n* are:

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

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

$$R_n = \frac{O_n}{O_1} = \frac{(n-1) + (n-2) + \ldots + \left(n - \frac{n}{2}\right)}{n} = \frac{n(\log n - 1) + 1}{n - 1} \approx \log n$$

8.10 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

$$S_P = T\,(s + q) / T(s + q / P) = 1 / (s + q / P)$$

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:

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

This is a line with a moderate slope.

8.11 Matrix-Vector Multiplication

In mathematical notation:

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \dots & \dots & \dots & \dots \\ A_{m1} & A_{m2} & \dots & A_{mn} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \dots \\ V_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} A_{1i}V_i \\ \sum_{i=1}^{n} A_{2i}V_i \\ \dots \\ \sum_{i=1}^{n} A_{mi}V_i \end{bmatrix}$$

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 represetned in Fortran 90 as follows:

```
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:

$$((s_* + n - 1)\tau + ((s_+ - 1)\log n + (n - 1))\tau)$$

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:

$$(m(\lceil \log n \rceil t_{+} + t_{*}))$$

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

```
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:

```
do j=1,n
   R(1:m)=R(1:m)+A(1:m,j)*V(j)
end do
```

The time for this loop in a pipelined machine is:

$$n(s_{+} + (m-1) + s_{*} + (m-1))\tau$$

if there is no chaining, and

$$n(s_{+} + s_{*} + (m - 1))\tau$$

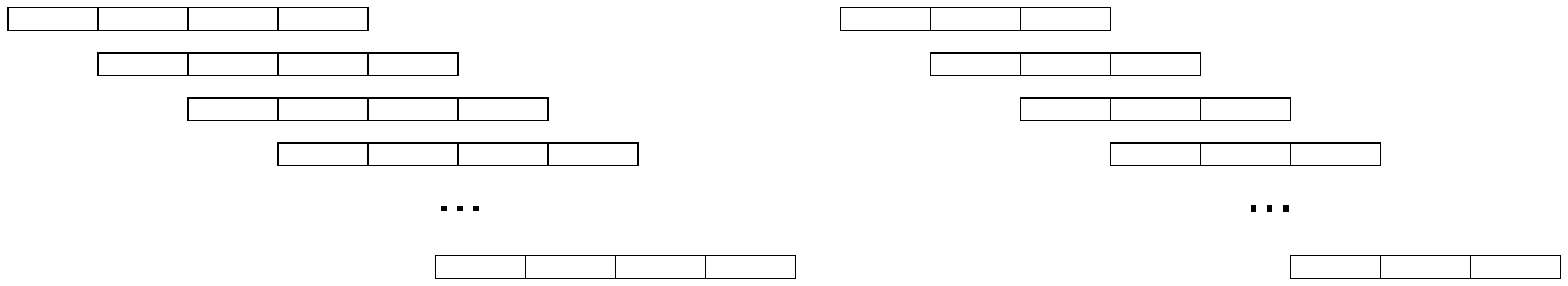
if there is chaining.

Assume two consecutive vector operations where the second operation uses the output from the first. *Chaining* allows the two operations to behave as if there were a single pipeline for both operations. This is achieved by feeding the output from the pipeline executing the first operation directly to the second pipeline. The alternative is to wait for the first opeation to complete before starting the second operation.

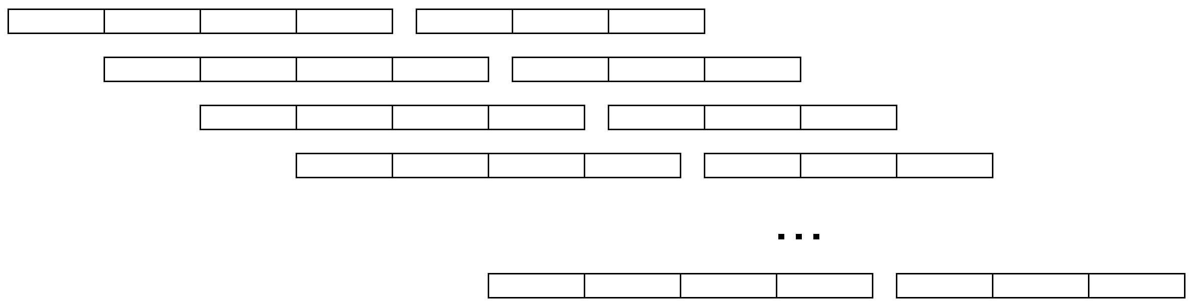
To illustrate chaining. consider the two statements:

```
a(1:n)=b(1:n)*c(1:n)
e(1:n)=a(1:n)+d(1:n)
```

Ignoring memory accesses and subscript computations, and assuming 4 stages for multiplication and 3 for addition, we have the following time lines in the absence of chaning:



But, in the presence of chaning, the time lines would look as follows:



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

$$(t_+ + t_*)n$$

8.12 Matrix Multiplication

1. *Inner product method.*

   Matrix multiplication is usually written:

```
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:

```
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:

$$(t_+\lceil \log n \rceil + t_*)n^2$$

2. *Middle-product method* (n-parallelism)

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

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

The direct translation of this loop into vector form is:

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

Alternatively, the headers could have been exchanged in a different order to obtain the loop:

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

The time on a pipelined machine, assuming chaining, is:

$$n^2(s_+ + s_* + (n-1))\tau$$

The time in an array machine is:

$$(t_+ + t_*)n^2$$

3. Outer-product method (n^2-parallelism)

Another interchange of the loop headers produce:

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

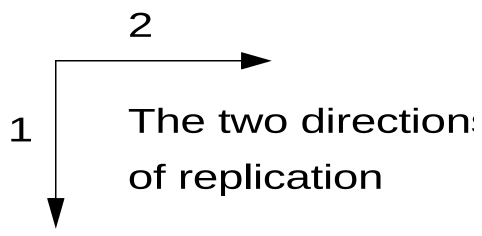
To obtain n^2 parallelism, the inner two loops should take the form of a matrix operation:

```
do k=1,n
   C(1:n,1:n)=C(1:n,1:n)+A(1:n,k)⊗ B(k,1:n)
end do
```

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:

$$\begin{bmatrix} A_{1k}B_{k1} & A_{1k}B_{k2} & A_{1k}B_{k3} & \dots \\ A_{2k}B_{k1} & A_{2k}B_{k2} & A_{2k}B_{k3} & \dots \\ A_{3k}B_{k1} & A_{3k}B_{k2} & A_{3k}B_{k3} & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$



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

$$\begin{bmatrix} A^k & A^k & \dots & A^k \end{bmatrix} \times \begin{bmatrix} \boldsymbol{B}^k \\ \boldsymbol{B}^k \\ \dots \\ \boldsymbol{B}^k \end{bmatrix}$$

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 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:

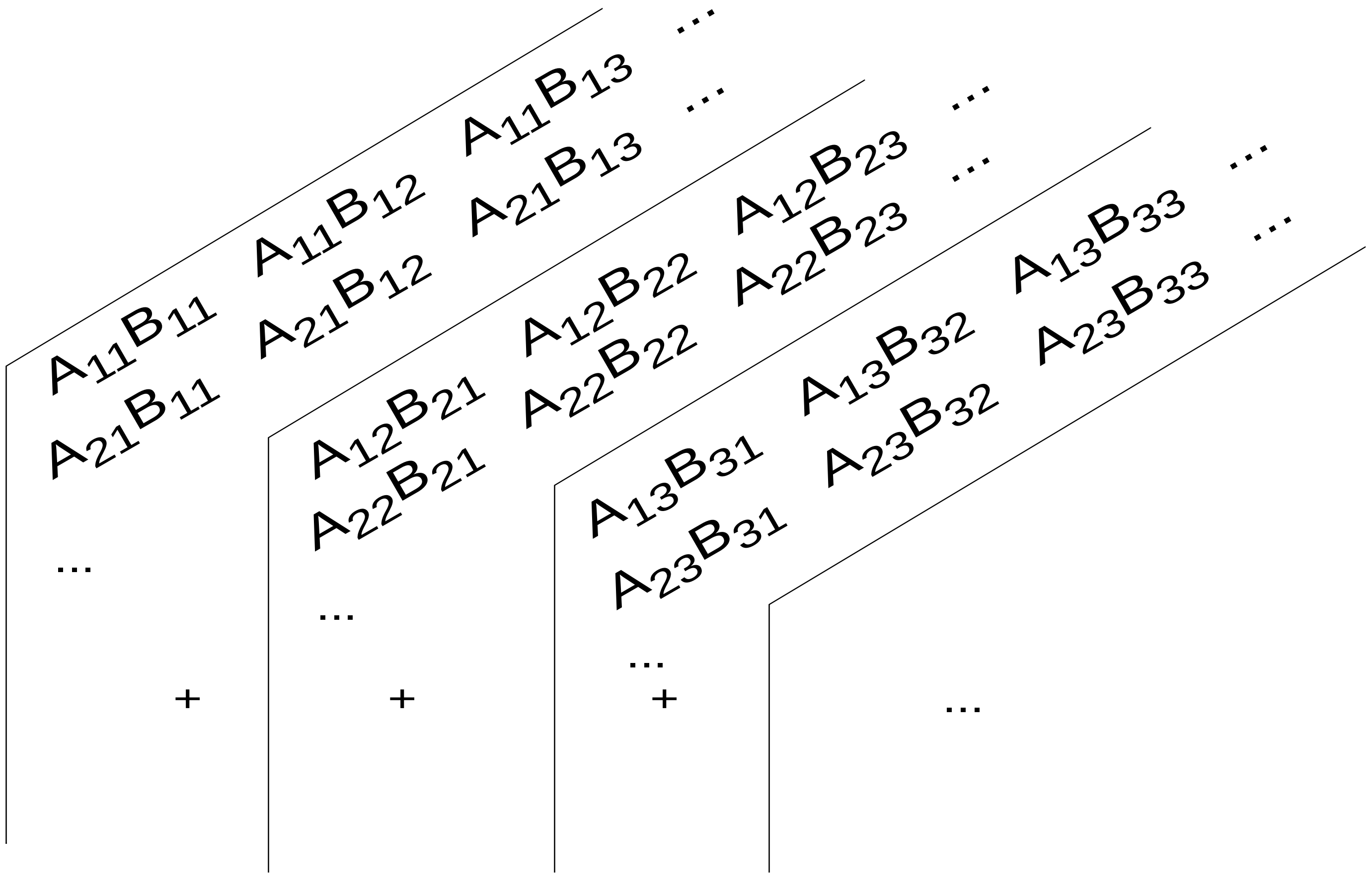
$$(2t_{\mathrm{copy}}\lceil \log n \rceil + t_{*} + t_{+}\lceil \log n \rceil)n$$

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×n matrices, `C=matmul(A,B)`, can be computed by adding n matrices of rank (n,n):

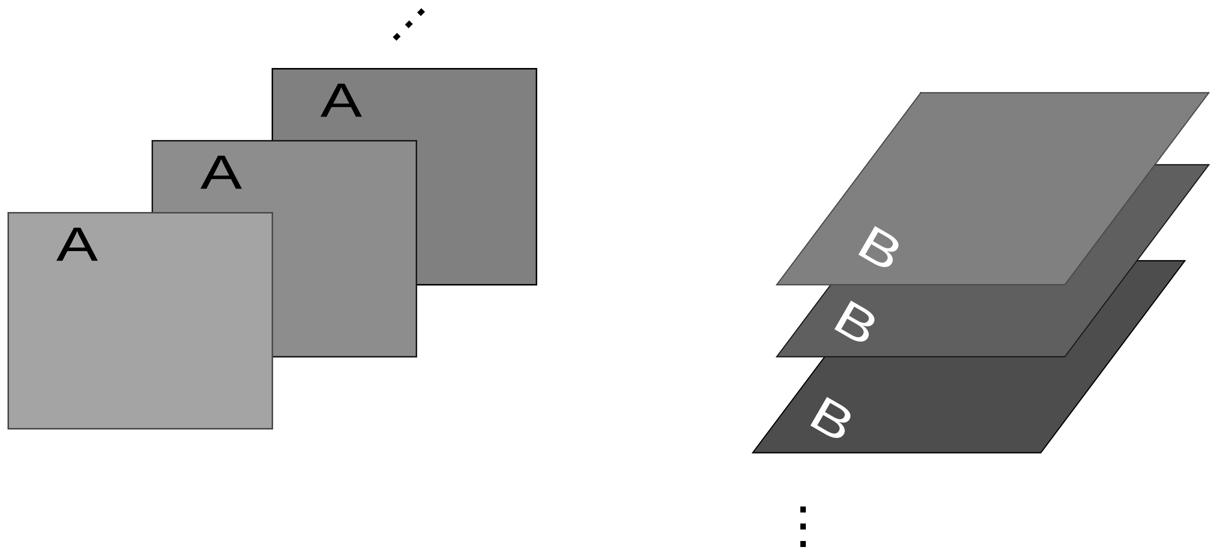
C =



$A_{11}B_{11}$ $A_{11}B_{12}$ $A_{11}B_{13}$ ...
$A_{21}B_{11}$ $A_{21}B_{12}$ $A_{21}B_{13}$ ...
...
+
$A_{12}B_{21}$ $A_{12}B_{22}$ $A_{12}B_{23}$ ...
$A_{22}B_{21}$ $A_{22}B_{22}$ $A_{22}B_{23}$ ...
...
+
$A_{13}B_{31}$ $A_{13}B_{32}$ $A_{13}B_{33}$ ...
$A_{23}B_{31}$ $A_{23}B_{32}$ $A_{23}B_{33}$ ...
...
+
...

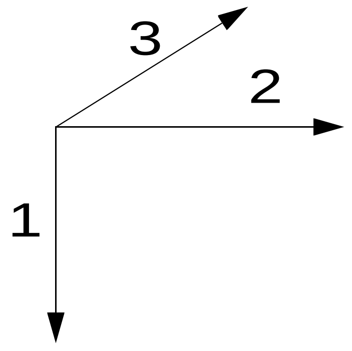
These n matrices of rank (n,n) can be computed by multiplying (element-by-element) two three-dimensional arrays of rank(n,n,n).

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



This replication can, again, be achieved, with SPREAD.

Thus, give the following three directions of replication:



+ \+

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

```
T(:,:,:)=SPREAD(A,DIM=3,NCOPIES=n)*SPREAD(B,DIM=1,NCOPIES=n)
```

Then, the result is just `C=SUM(T,DIM=2)`

In an array machine with P>=n^3 processing unit, the time to compute C would be:

$$(2t_{\text{copy}}\lceil \log n \rceil + t_* + t_+)$$

8.13 Multiplication by Diagonals

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

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

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:

$$\begin{bmatrix} A_0 & A_1 & \dots & A_{\beta_2} & 0 & 0 & 0 \\ A_{-1} & \dots & \dots & \dots & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ A_{-\beta_1} & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \dots & \dots & \dots \end{bmatrix} \times V$$

After separating the diagonals into separate matrices, we get:

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

which can be written as follows:

$$A_0 V \overset{\wedge}{+} A_1 V^2 \overset{\wedge}{+} \cdots \overset{\wedge}{+} A_{\beta_2} V^{\beta_2} + A_{-1} V_{n-1} \overset{+}{\vee} \cdots \overset{+}{\vee} A_{-\beta_1} V_{n-\beta_1}$$

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):

```
    A0(1:n)*V(1:n)  +
(/  A1(1:n-1)*V(2:n),0. /)  +
...
(/  Aβ1(1:n-β1)*V(β1+1:n), (0., j=1,β1) /) +
(/  0., A-1(1:n-1)*V(1:n-1) /) +
...
(/ (0., j=1,β2), Aβ2(1:n-β2)*V(1:n-β2) /)
```

8.14 Consistent Algorithms

A vector algorithm for solving a problem of size *n* is consistent with the best serial algorithms for the same problem if the redundancy is bounded as n $\rightarrow \infty$.

Vector reduction is consistent.

$$R_n = \frac{O_n}{O_1} = \frac{\frac{n}{2} + \frac{n}{4} + \ldots + 1}{n-1} = \frac{n-1}{n-1} \rightarrow 1$$

But parallel prefix is not (see Section 8.9).

The time for the simple form of parallel prefix presented in Section 8.7 given P processing units and vector length $n=2^k$ is:

$$t_{parallel} = \sum_{i=0}^{k-1} \left\lceil \frac{n-2^i}{P} \right\rceil$$

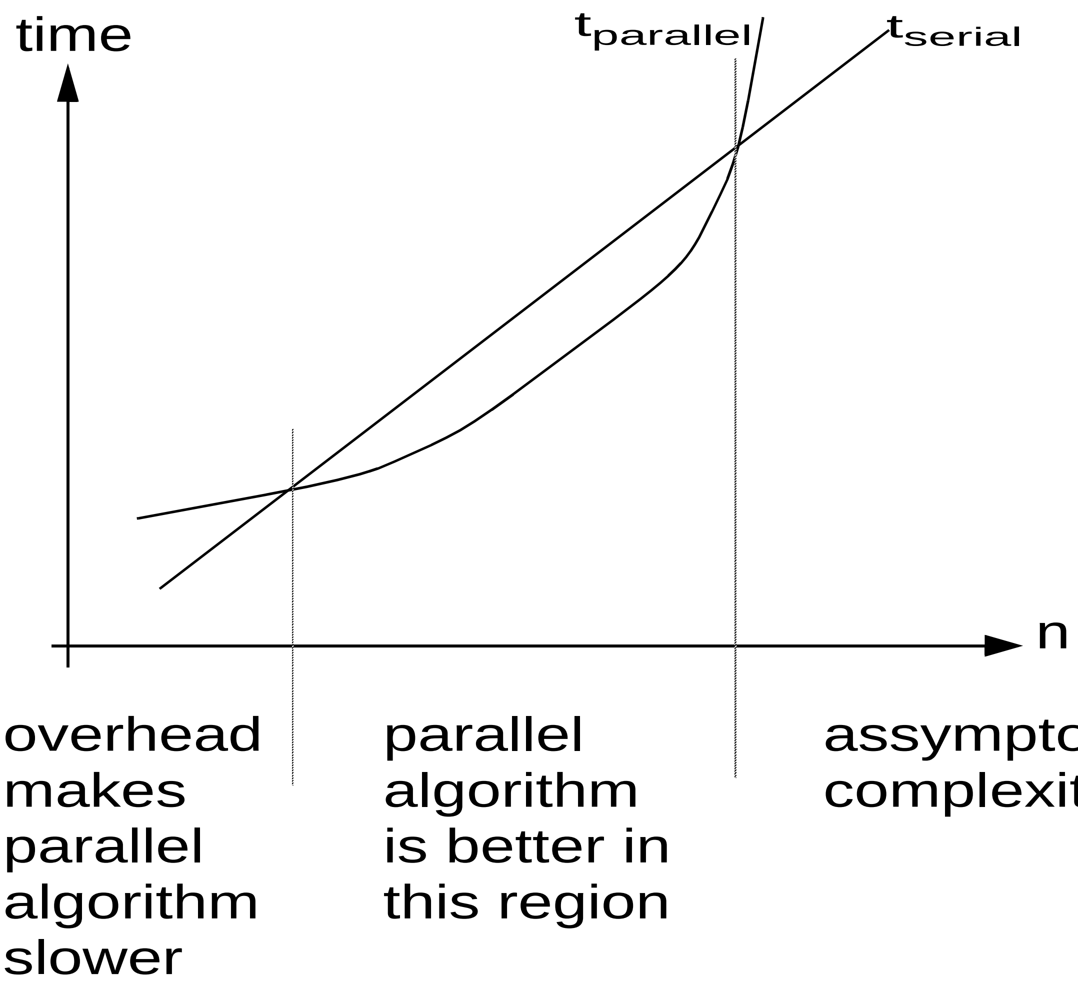
The problem with algorithms that are non consistant is that for n (size of input data) large enough, the array algorithm is slower than the scalar version. Assuming constant number of devices (pipeline stages or processing units).

For example, assuming 8 processors $t_{parallel}$ has the following values:

n	t_{serial}	$t_{parallel}$
16	15	7
32	31	17
64	63	41
512	511	513
1024	1023	1153
$\sim 10^6$	$\sim 10^6$	$\sim 2.5 \times 10^6$

Notice that the array algorithm becomes slower than the scalar version when n=512.

A typical situation with inconsistent algorithms is depicted in the following graph:



8.15 A consistent version of parallel prefix

A consistent version of parallel prefix can be obtained by blocking the original algorithm. This can be done for both a pipelined unit or an array machine. We will show the array machine version. There are several steps in the algorithm

1. First the array B(1:n) should be reshaped into an $\left\lceil \frac{n}{P} \right\rceil \times P$ matrix as follows:

   $$\texttt{C=RESHAPE(B,(/ } \left\lceil \frac{n}{P} \right\rceil \texttt{ ,P /),0.0)}$$

   The `reshape(source,shape[,pad][,order])` function takes the elements of `source`, in normal Fortran order, and returns them (as many as will fit) as an array whose shape is specified by the one-dimensional integer array `shape`. If there is space remaining, then `pad` must be specifie and is used to fill out the rest. See the manual for a description of `order`.

   Thus, if

```
B = (/1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,11. /)
```

then, the result of RESHAPE above would be:

$$\begin{bmatrix} 1. & 5. & 9. \\ 2. & 6. & 10. \\ 3. & 7. & 11. \\ 4. & 8. & 0. \end{bmatrix}$$

2. Then, we assign a column of C to each processing unit and compute the partial sums of each column separately as follows:

```
do i=2, $\left\lceil \frac{n}{P} \right\rceil$
    C(i,1:P) = C(i-1,1:P) + C(i,1:P)
end do
```

At the end of this loop we will have

$$C(k,q) = \sum_{i=1}^{k} C(i,q)$$

3. The third step is to compute the true value of the last element of each column:

```
do j=2,P
    C( $\left\lceil \frac{n}{P} \right\rceil$ ,j)= C( $\left\lceil \frac{n}{P} \right\rceil$ ,j-1)+C( $\left\lceil \frac{n}{P} \right\rceil$ ,j)
end do
```

At the end of this step, the last element of each column will have the sum of all elements in its own column and the columns preceding it.

4. Finally, the elements in each processor are adjusted:

```
do i=1, $\left\lceil \frac{n}{P} \right\rceil$ -1
    C(i,2:P) = C( $\left\lceil \frac{n}{P} \right\rceil$ ,1:P-1) + C(i,2:P)
end do
```

The total number of operation in the algorithm is:

$$\left(\left\lceil \frac{n}{P} \right\rceil - 1\right)P + (P-1) + \left(\left\lceil \frac{n}{p} \right\rceil - 1\right)(P-1)$$

The algorithms is consistent because:

$$\frac{\left(\left\lceil \frac{n}{P} \right\rceil - 1\right)P + (P-1) + \left(\left\lceil \frac{n}{p} \right\rceil - 1\right)(P-1)}{n} \rightarrow 2$$

8.16 Cyclic Reduction

Let

$$S_1 = a_1 S_0 + b_1$$

$$S_i = a_i S_{i-1} + b_i \qquad i = 2, \ldots, n$$

One way to solve the recurrence in parallel is to use cyclic reduction:

From

$$S_i = a_i S_{i-1} + b_i$$

and

$$S_{i-1} = a_{i-1} S_{i-2} + b_{i-1}$$

we get

$$S_i = a_i a_{i-1} S_{i-2} + a_i b_{i-1} + b_i$$

Which can be rewritten as

$$S_i = a_i^{(1)} S_{i-2} + b_i^{(1)}$$

where $a_i^{(1)}$ and $b_i^{(1)}$ are defined as follows:

$$a_i^{(1)} = a_i a_{i-1}$$

$$b_i^{(1)} = a_i b_{i-1} + b_i$$

Now we have S_i as a function of S_{i-2}.

If we repeat this process several times we obtain

$$S_i = a_i^{(l)} S_{i-2^l} + b_i^{(l)}$$

$$l = 0, 1, \ldots, \log n$$

$$i = 1, 2, \ldots, n$$

where

$$a_i^{(l)} = a_i^{(l-1)} a_{i-2^{l-1}}^{(l-1)}$$

$$b_i^{(l)} = a_i^{(l-1)} b_{i-2^{l-1}}^{(l-1)} + b_i^{(l-1)}$$

Initially,

$$a_i^{(0)} = a_i$$

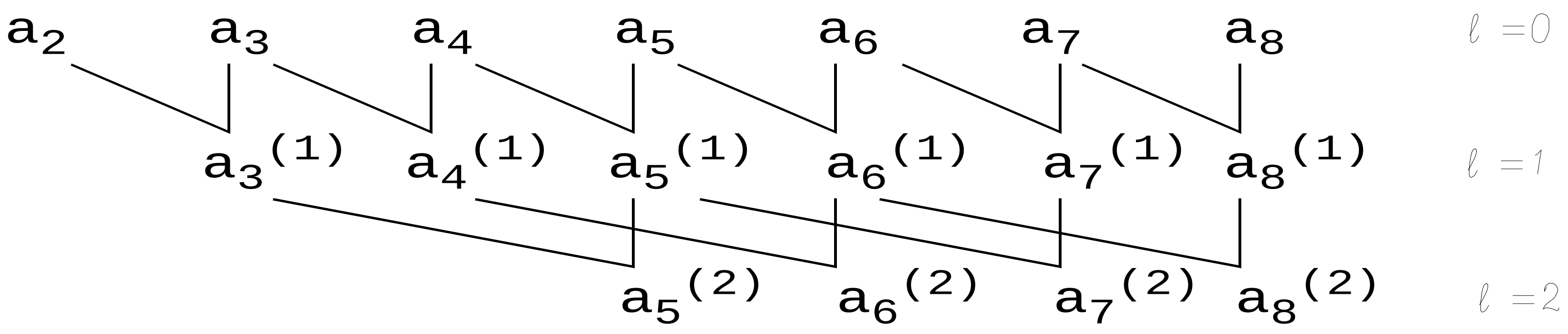
$$b^{(0)} = b_i$$

When the subscript of a_i, b_i or S_i is outside the range *1,...,n*, the value *0* should be assumed

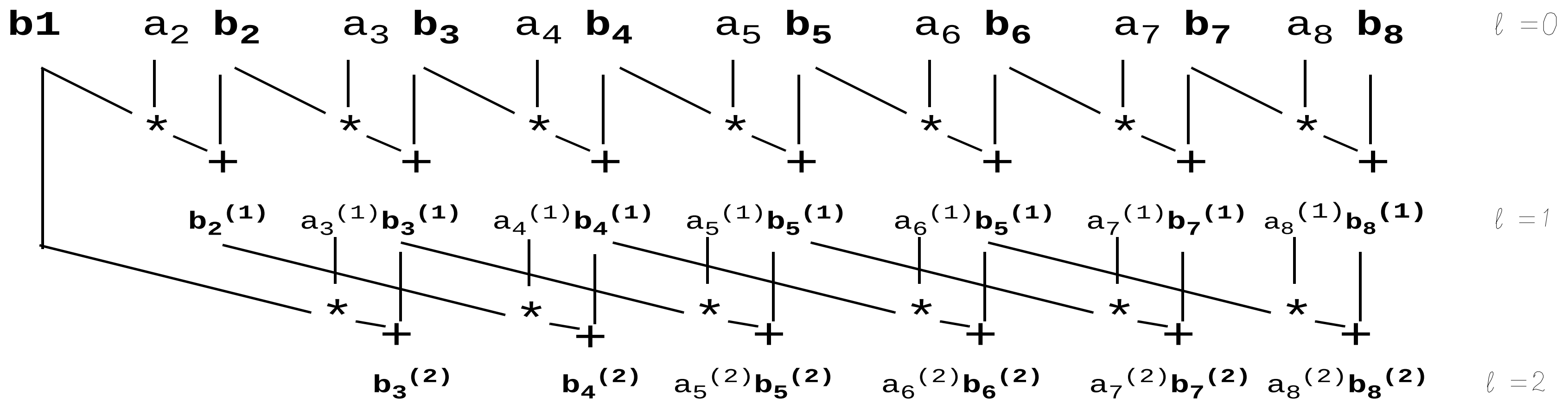
When $l = \log n$

$$S_i = 0 + b_i^{(\log n)}$$

To compute the *a*'s and *b*'s in parallel we proceed as shown below



a_2 a_3 a_4 a_5 a_6 a_7 a_8
$\ell = 0$
$a_3^{(1)}$ $a_4^{(1)}$ $a_5^{(1)}$ $a_6^{(1)}$ $a_7^{(1)}$ $a_8^{(1)}$
$\ell = 1$
$a_5^{(2)}$ $a_6^{(2)}$ $a_7^{(2)}$ $a_8^{(2)}$
$\ell = 2$



The resulting program in Fortran 90 is

```
S(1:n)=b(1:n)
do i=1,logn
   S(1:n)=EOSHIFT(S(1:n),-2**(i-
1))*a(1:n)+S(1:n)
   a(1:n)=EOSHIFT(a(1:n),-2**(i-1))*a(1:n)
end do
```

The function `eoshift(array,shift [,boundary] [,dim])` returns the result of end-off left-shifting every one-dimensional section of `array` (in dimension `dim`) by `shift`. If `shift` is negative, the array is shifted to the right. If `boundary` is present as a scalar, it supplies the elements to fill in the blanks; if it is not present, zero is used. If `dim` is not present, one is assumed.

8.17 The `FORALL` statement

“One of the positive effectns of Fortran 90’s long gestation period has been the general recognition, both by the X3J3 committee and by the community at large, that Fortran needs to evlove over time. Fortran 95 is a minor, but by no means insginificant, updating of Fortran 90.”

It includes `PURE` and `ELEMENTAL` procedures, the `DIM` argument for `maxloc` and `minloc`, and the `FORALL` statement.

“Because the inside iteration of a `FORALL` block can be done in any order, or in parallel, there is a logical difficulty in allowing functions or subroutines inside such blocks: If the function or subroutine has *side effects* (that is, if it changes any data elsewhere in the machine, or in its own saved variables) then the result of a `FORALL` calculation could depend on the order in which the iterations happen to be done. This can’t be tolerated, of course; hence a new `PURE` attribute for subprograms.”

From *W. H. Press et al.*

The `FORALL` statement is used to specify an array assignment in terms of individual elements or sections.

For example,

```
      A(1:10,2:20)=5*B(2:11,1:19)
```

can al be written as follows:

```
      forall(i=1:10,j=1:19)A(i,j+1=5*B(i+1,j)
```

Notice that `FORALL` specifies a vector operations and therefore, the right-hand side is evaluated before any part of the left-hand side is changed.

Thus, the previous assignment can be written as:

```
         do i=1,10
            do j=1,19
               A(i,j+1)=5*B(i+1,j)
            end do
         end do
```

because the right and left-hand sidesare disjoint, but

```
      forall (j=1:n) A(j)=5*A(j-1)
```

should be written as two loops:

```
      do j=1,n
         temp(j)=5*A(j-1)
      end do
      do j=1,n
         A(j)=temp(j)
      end do
```

A singlle loop would not do in this case.

Notice that sometimes a single loop would suffice even if there is overlap betwenn left- and right-hand sides. Thus,

```
      forall (j=1:n) A(j)=5*A(j+1)
```

Is equivalento to the loop:

```
      do j=1,n
         A(j)=5*A(j=1)
      end do
```

Several examples of FORALL are presented next:

1.
```
FORALL(I=1:10) A(I,I)=B(I,I)
```

can be written as
```
        A(1:10)=B(1:10)
```

or as
```
        A=B
```

if both A and B have shape (10).

2.
```
FORALL(I=1:10:2,J=10:1:-1)A(I,J)=B(I,J)*C(I,J)
```

can be written as
```
    A(1:10:2,10:1:-1)=B(1:10:2,10:1:-1)*C(1:10:2,10:1:-1)
```

3.
```
FORALL(I=1:10)A(I)=I
```

Is equivalent to
```
        A(1:10)=(/1:10/)
```

4.
```
FORALL(I=1:10,J=1:20)A(I,J)=B(I)
```
Needs SPREAD to be implemented because array sections in an assignment statement have to be comformable.

```
      A=SPREAD(B,DIM=2,NCOPIES=20).
```

```
5.    FORALL(I=1:10,J=1:10)A(I,J)=B(J,I)
```

Can be implemented with the `TRANSPOSE` intrinsic functions.

```
6.    FORALL(I=1:N) A(I,I)=B(I,I)
```

or the slightly more complex statement:

```
      FORALL(I=1:N) A(I,I)=B(I+1,I-1)
```

would need `RESHAPE` (or `EQUIVALENCE`) to be written as vector operations is Fortran 90. Thus, the first `FORALL` could be written as:

```
      T1(1:N*N)=RESHAPE(B,(\N*N\))
      T2(1:N*N)=RESHAPE(A,(\N*N\))
      T2(1:N*N:N+1)=T1(1:N*N:N+1)
      A=RESHAPE(T2,(\N,N\))
```

The second `FORALL` is left as an excercise.

```
7. FORALL(I=1:N,J=1:N,K=1:N,
   &      I+J+K.EQ.3*(N+1)/2)
   &              A(I+J-K,J)=B(I,J,K)
```

This statement includes a boolean expression. Only for those elements where the expression is true, an assignment will take place.

```
8. FORALL(I=1:10,J=1:20,K=1:30)
   &              A(I,J,K)=I+J+K
```

For this statement SPREAD is needed:

```
   A=SPREAD(SPREAD((/1:10/),2,20),3,30)
  &  +SPREAD(SPREAD((/1:20/),1,10),3,30)
  &  +SPREAD(SPREAD(/1:30/),1,10),2,20)
```

```
9. FORALL(I=2:2000)A(I)=A(I/2)
```

This pattern is a standard technique for representing a binary tree structure as an array; the two children of of element *k* are elements *2k* and *2k+1*. This statment causes every node in the tree to receive a copy of information from its parent; it might be part of a computation that pipelines data down the leaves of the tree in a breadth-first fashion.

```
10.FORALL(I=1:10,J=1:10,K=1:10)
   &           A(I,J,K)=B(J,K,I)
```

This particularly simple example, can be expressed with the RESHAPE intrinsic, which provides for permuting axes while reshaping.

```
11.FORALL(I=1:10,J=1:10,K=1:10)
   &          A(I,J,K) = B(J,11-K,I+1)
```

In this case one would have to use RESHAPE to transpose the axes and then use an array section assignment to do the rest of the job. This would be relatively inefficent, resulting in multiple copying of data.

8.18 Sorting in Fortran 90.

There are many parallel sorting algorithms. We will discuss two very simple ones in this chapter and more elaborate algorithms later in the semester.

Perhaps the simplest sorting algorithm is *bubble sort*. (Text extracterd from Kumar et al. *Introduction to Parallel Computing*) It compares and exchages adjacent elements in the sequence to be sorted. Given the sequence $a_1, a_2, ..., a_n$, the algorith first performs n-1 compare-exchange operations in the following order: $(a_1,a_2),(a_2,a_3), ...(a_{n-1},a_n)$. This step moves the largest element to the end of the sequence. The last ement in the sequence is then ignored, and the sequence of compare exchanges is applied to the resulting sequence. The sequence is sorted after *n-1* iterations. The algorithm is as follows:

```
do i=n-1,1,-1
   do j=1,i
      if (a(j) > a(j+1) ) swap (a(j),a(j+1))
   end do
end do
```

Where *swap(a,b)* is just the sequence

```
      t=a
      a=b
      b=t
```

This algorithm can be easily parallelized as discussed later on.

For vectorization, we will use the following slightly modifed version known as *odd-even transposition*:

```
do i=1,n
   if i is odd then
      do j=0,n/2-1
         if (a(2*j+1)>a(2*j+2)) swap(a(2*j+1),a(2*j+2))
      end do
   end if
   if i is even then
      do j=1,n/2-1
         if (a(2*j)>a(2*j+1)) swap(a(2*j),a(2*j+1))
      end do
   end if
end do
```

The algorithm alternates between two phases: odd and even. During the odd phase, elements with odd indices are compared with their right neighbors, and if they are out of sequence they are exchanged. Similarly, during the even phase, elements with even indices are compared with their right neighbors, and if they are out of sequence they are exchanged.

Vectorization is quite simple:

```
do i=1,n
   if i is odd then
      where (a(1:n-1:2)>a(2:n:2))
         swap (a(1:n-1:2),a(2:n:2))
      end where
   end if
   if i is even then
      where (a(2:n-2:2)>a(3:n-1:2))
         swap (a(2:n-2:2),a(3:n-1:2))
      end where
   end if
end do
```

Bubble sort is not a very efficient algorithm. It takes *n(n-1)/2* comparisons to complete. The parallel version reduces that to *n* steps, but a good sequential algorithm only requires a number of comparisons proportional to *n log n*. And there are paralle algorithm that require time proportional to $\log^2 n$. So this is ok, but not great.

A better sorting algorithms in some situations is *radix sort*. This was the algorithm used to sort punched cards with electro-mechanical devices.

The idea is that the values to be sorted are assumed to be numbers in a certain radix. Integers could be radix 10 or 2 depending on the circumstances. For punched cards, it was base 10. In today's machines, we could assume base two, but any other base can be assumed. When values are names, base 26 can be assumed.

Radix sort, goes through all the "digits" starting with the less significant one. For each digit it processes the whole sequence. Elements of the sequence are placed in separate buckets, one for each possible digit. Placement in the buckets is in the order theelements appear in the sequence. After processing all elements for a particular position, the buckets are catenated to create the sequence for the next position.

Consider for example the following sequence:

223, 148, 221, 071, 138, 131.

After the first step, the sequence will be separated as follows:

bucket	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
		221		*223*					*148*	
		071							*138*	
		131								

After catenation, we get: *221,071,131,223,148,138*.

Now, the digits in the second position are processed:

bucket	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
			221	*131*	*148*			*071*		
			223	*138*						

Again,the buckets are catenated: *221,223,131,138, 148,071.*

Then, the digits in the third position are processed:

bucket	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	071	*131*	*221*							
		138	*223*							
		148								

Finally, the sorted sequence is obtained by catenating the buckets: *071,131,138,148, 221, 223.*

The algorithm can be easily implemented in Fortran 90 using the `pack` intrinsic function. `Pack(array, mask)` returns a one-dimensional array containing the elements of `array` to pass the `mask`.

Thus, assuming that the sequence to be sorted is in vector a, and that the elements are in base *b* and contain *d* digits each, we can proceed as follows:

```
do i=1,d
   m_0(1:n) = the digit in a(1:n) with weight $b^{i-1}$ is 0
   m_1(1:n) = the digit in a(1:n) with weight $b^{i-1}$ is 1
   ...
   m_b1(1:n) = the digit in a(1:n) with weight $b^{i-1}$ is b-1
   a=(/pack(a,m_0),pack(a,m_1),...,pack(a,m_b1)/)
end do
```

In particular for base 2, only one mask is needed:

```
do i=1,d
   m=mod(a,2**i)<2**(i-1)
   a= (/pack(a,m),pack(a,.not.m)/)
end do
```

Pack can be implemented in parallel using the primitives discussed earlier in class:

```
function pack(a,m)

   where (m)
      c=1
   elsewhere
      c=0
   end where
   order=parallel_prefix(c)
   where (m)
      temp(order)=a
   end where
   pack=temp
   return
end
```

8.19 Processing linked lists

Lined lists are usually represented with pointer. Pointer values are usually memory addresses. However, linked lists can also be represented using array locations.

For example, a linked list could be represented using two arrays. One containing the `value` of the list entry, and the other conaining the position within the array where the `next` element in the list is located.

Now, a vector algorithm to make vector `next` point to the last entry in the list is as follows[1]:

```
do while (any(next/=null).and.any(next(next)/=null))
   where(next(next)/=null)
      next=next(next)
   end where
end do
```

1.Here we assume that `next(null)=null`

And a vector algorithm to do parallel prefix computation in the order of the linked list is as follows:

```
do while (any(next /= null))
   where (next /= null)
      value(next)=value+value(next)
      next=next(next)
   end where
end do
```

8.20 The Wavefront method (a.k.a. the Hyperplane method)

In this chapter we will only consider a simple two dimensional Fortran 77 form of this method. That is, we will only consider two dimensional loop nests with a single statement inside that assigns to a two dimensional array.

To illustrate this method we will draw a graph of the iteration space of the loop. Each iteration will be a node in the graph. The graph will taked the form of a mesh with equal vertical and horizontal separation.

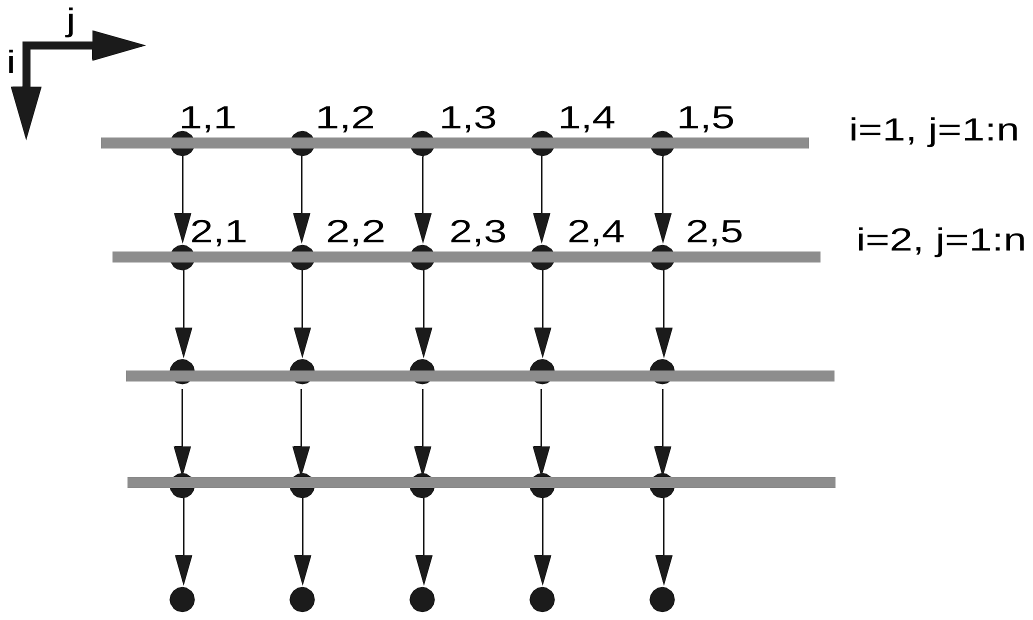
These nodes will be joined by three classes of arcs representing races (write-read, read-write, write-write). These arcs (which are called **dependences**) will always flow in the direction of execution in the original loop.

The idea is that a vector form can be obtained by finding a collection of parallel lines that are equidistant, are not parallel to any dependence arc, and pass through all the nodes in the graph.

For example, the loop

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

can be represented by the following graph:

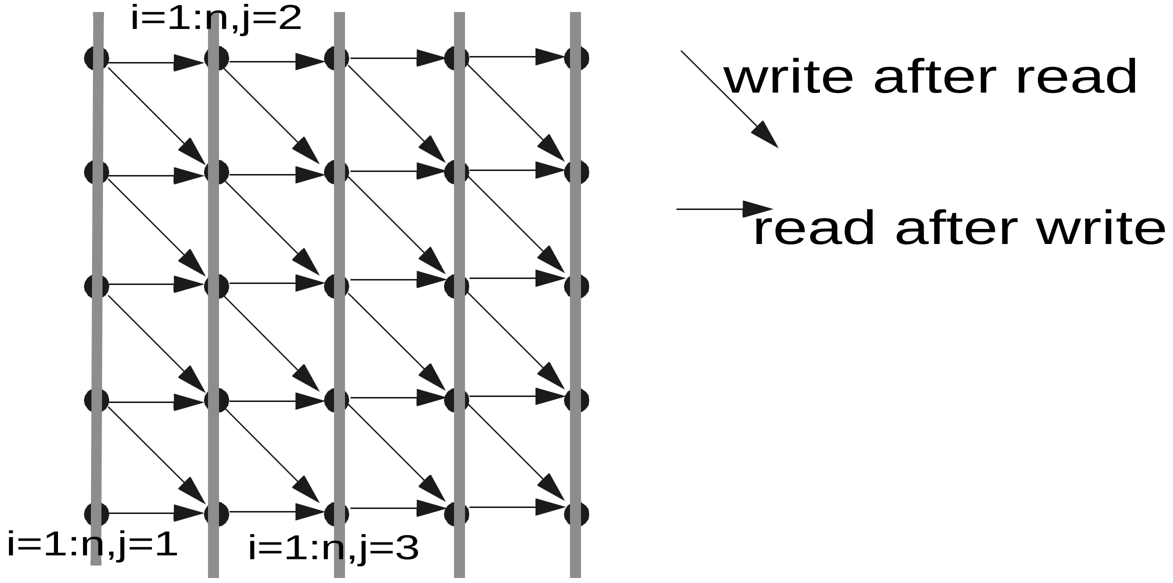


From the graph it is clear that for each i there is a vector operation in j.

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

A second example:

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

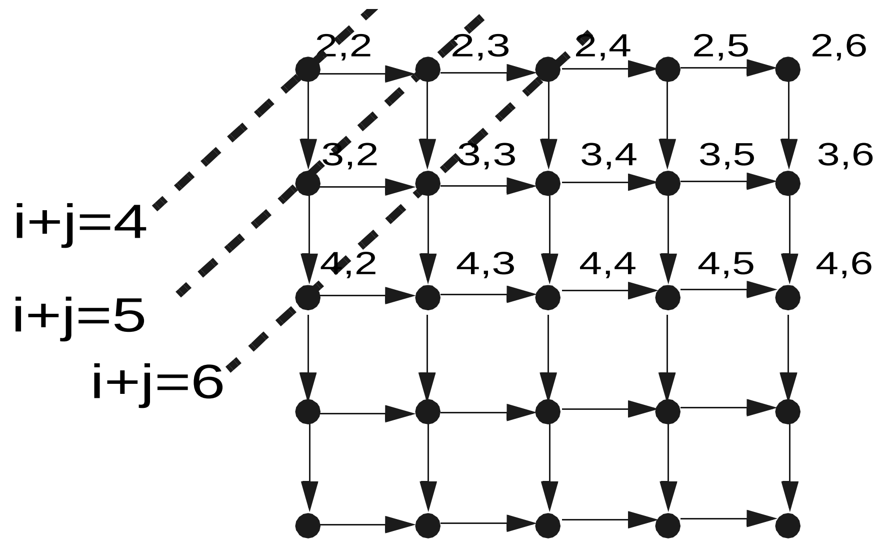


Now for each j there is a vector operation

```
do j=1,n
   a(1:n,j)=a(1:n,j-1)+a(2:n+1,j+1)+b(i)+c(1:n)
end do
```

A more complicated case:

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



From the equations:

$$2 \le i \le n$$
$$2 \le k - i \le n$$
$$k = 4, 5, \ldots, 2n$$

We conclude that:

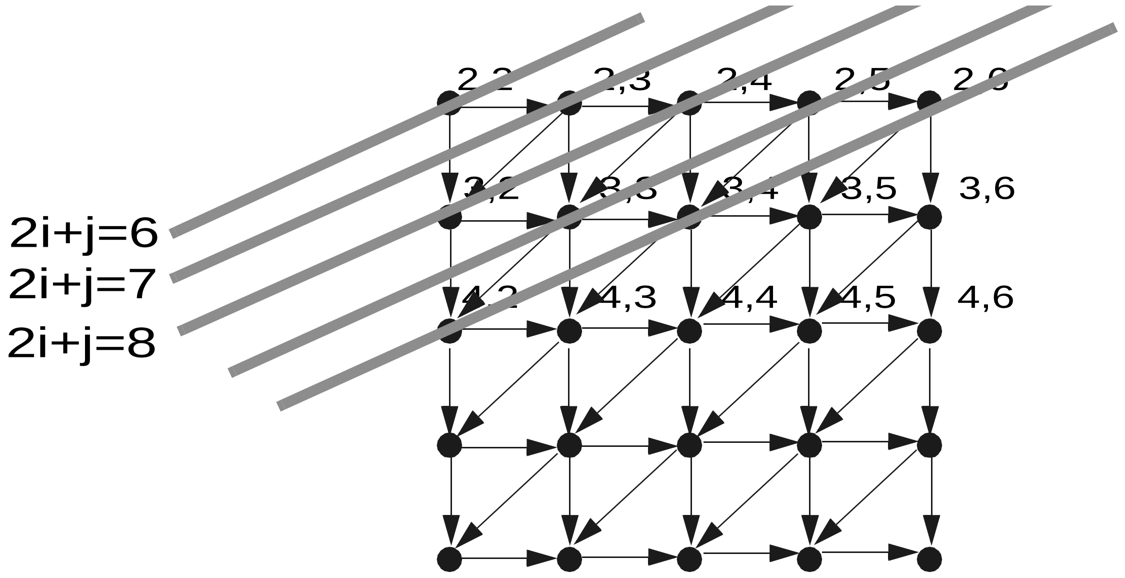
$$max(2, k-n) \le i \le min(n, k-2)$$

From where:

```
do k=4,2*n
   forall (i=max(2,k-n):min(n,k-2)) a(i,k-j)=...
end do
```

Another complex example:

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



From the equations:

$$4 \le 2i \le 2n$$
$$2 \le k - 2i \le n$$
$$k = 6, 5, \ldots, 3n$$

We conclude that:

$$max\left(2, \left\lceil \frac{k-n}{2} \right\rceil\right) \le i \le min\left(n, \left\lfloor \frac{k-2}{2} \right\rfloor\right)$$

From where:

```
do k=6,3*n
   forall (i=max(2,(k-n+1)/2):min(n,(k-2)/2)) a(i,k-j)=...
end do
```