8.16 Cyclic Reduction

Let

\[ S_1 = a_1 S_0 + b_1 \]

\[ S_i = a_i S_{i-1} + b_i \quad i = 2, \ldots, n \]

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

\[ S_i = a_i^{(l)} S_{i-2}^l + b_i^{(l)} \]

\[ l = 0, 1, ..., \log n \]

\[ i = 1, 2, ..., n \]
where

\[ a_i^{(l)} = a_i^{(l-1)} a_{i-2}^{(l-1)} \]

\[ b_i^{(l)} = a_i^{(l-1)} b_{i-2}^{(l-1)} + b_i^{(l-1)} \]

Initially,

\[ a_i^{(0)} = a_i \]

\[ b_i^{(0)} = b_i \]

When the subscript of \( a_i, b_i \) or \( S_i \) is outside the range \( 1, \ldots, n \), the value 0 should be assumed.
For the case $n=8$, the sequence of substitutions is as follows:

\[
\begin{align*}
S_1 &= a_1 S_0 + b_1 \\
S_2 &= a_2 S_1 + b_2 \\
S_3 &= a_3 S_2 + b_3 \\
S_4 &= a_4 S_3 + b_4 \\
S_5 &= a_5 S_4 + b_5 \\
S_6 &= a_6 S_5 + b_6 \\
S_7 &= a_7 S_6 + b_7 \\
S_8 &= a_8 S_7 + b_8
\end{align*}
\]
\[
\begin{align*}
S_1 &= a_1^{(1)} S_0 + b_1^{(1)} \\
S_2 &= a_2^{(1)} S_0 + b_2^{(1)} \\
S_3 &= a_3^{(1)} S_1 + b_3^{(1)} \\
S_4 &= a_4^{(1)} S_2 + b_4^{(1)} \\
S_5 &= a_5^{(1)} S_3 + b_5^{(1)} \\
S_6 &= a_6^{(1)} S_4 + b_6^{(1)} \\
S_7 &= a_7^{(1)} S_5 + b_7^{(1)} \\
S_8 &= a_8^{(1)} S_6 + b_8^{(1)}
\end{align*}
\]
\[
\begin{align*}
S_1 &= a_1^{(2)} S_0 + b_1^{(2)} \\
S_2 &= a_2^{(2)} S_0 + b_2^{(2)} \\
S_3 &= a_3^{(2)} S_0 + b_3^{(2)} \\
S_4 &= a_4^{(2)} S_0 + b_4^{(2)} \\
S_5 &= a_5^{(2)} S_1 + b_5^{(2)} \\
S_6 &= a_6^{(2)} S_2 + b_6^{(2)} \\
S_7 &= a_7^{(2)} S_3 + b_7^{(2)} \\
S_8 &= a_8^{(2)} S_4 + b_8^{(2)} \\
\end{align*}
\]
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

- $l = 0$
  - $a_2$
  - $a_3^{(1)}$
  - $a_4^{(1)}$
  - $a_5^{(1)}$
  - $a_6^{(1)}$
  - $a_7^{(1)}$
  - $a_8^{(1)}$

- $l = 1$
  - $a_3^{(2)}$
  - $a_4^{(2)}$
  - $a_5^{(2)}$
  - $a_6^{(2)}$
  - $a_7^{(2)}$
  - $a_8^{(2)}$

- $l = 2$
The resulting program in Fortran 90 is

\[
S(1:n) = b(1:n) \\
do\ i=1,\log n \\
\quad S(1:n) = \text{EOSHIFT}(S(1:n), -2^{(i-1)}) \ast a(1:n) + S(1:n) \\
\quad a(1:n) = \text{EOSHIFT}(a(1:n), -2^{(i-1)}) \ast a(1:n) \\
end\ do
\]
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 effects 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 evolve over time. Fortran 95 is a minor, but by no means insignificant, 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 all be written as follows:

$$\text{forall}(i=1:10,j=1:19)A(i,j+1=5*B(i+1,j)$$

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:

```plaintext
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 sides are disjoint, but

$$\text{forall } (j=1:n) \ A(j)=5*A(j-1)$$

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 single loop would not do in this case.

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

$$\text{forall } (j=1:n) \ A(j)=5*A(j+1)$$

Is equivalent 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 must 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:

\[
\begin{align*}
T1(1:N*N) &= \text{RESHAPE}(B, (\ N*N\ )) \\
T2(1:N*N) &= \text{RESHAPE}(A, (\ N*N\ )) \\
T2(1:N*N:N+1) &= T1(1:N*N:N+1) \\
A &= \text{RESHAPE}(T2, (\ N,N\ ))
\end{align*}
\]

The second FORALL is left as an excercise.
7. \texttt{FORALL (I=1:N, J=1:N, K=1:N,}
& \texttt{ I+J+K.EQ.3*(N+1)/2)}
& \texttt{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. \texttt{FORALL (I=1:10, J=1:20, K=1:30)}
& \texttt{A(I,J,K)=I+J+K}

For this statement \texttt{SPREAD} is needed:

\texttt{A=SPREAD(SPREAD((/1:10/),2,20),3,30)}
& \texttt{SPREAD(SPREAD(1:20/),1,10),3,30)}
& \texttt{SPREAD(SPREAD(/1:30/),1,10),2,20)}

9. \texttt{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 element \texttt{k} are elements \texttt{2k} and \texttt{2k+1}. This statement 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 inefficient, 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 extracted from Kumar et al. Introduction to Parallel Computing)

It compares and exchanges adjacent elements in the sequence to be sorted. Given the sequence $a_1, a_2, \ldots, a_n$, the algorithm first performs $n-1$ compare-exchange operations in the following order: $(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$. This step moves the largest element to the end of the sequence. The last element 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:

\[
\begin{align*}
\text{do } & i=n-1,1,-1 \\
& \quad \text{do } j=1,i \\
& \quad \quad \text{if } (a(j) > a(j+1) ) \quad \text{swap } (a(j),a(j+1)) \\
& \quad \text{end do} \\
& \text{end do}
\end{align*}
\]
Where \( swap(a,b) \) is just the sequence

\[
\begin{align*}
t &= a \\
a &= b \\
b &= t
\end{align*}
\]

This algorithm can be easily parallelized as discussed later on.

For vectorization, we will use the following slightly modified version known as \textit{odd-even transposition}:

\[
\text{do } \ i=1,n \\
\quad \text{if } i \text{ is odd } \text{ then} \\
\quad \quad \text{do } j=0,n/2-1 \\
\quad \quad \quad \text{if } (a(2j+1)>a(2j+2)) \text{ swap}(a(2j+1),a(2j+2)) \\
\quad \quad \text{end do} \\
\quad \text{end if} \\
\quad \text{if } i \text{ is even } \text{ then} \\
\quad \quad \text{do } j=1,n/2-1 \\
\quad \quad \quad \text{if } (a(2j)>a(2j+1)) \text{ swap}(a(2j),a(2j+1)) \\
\quad \quad \text{end do} \\
\quad \text{end if} \\
\text{end do}
\]

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:

```plaintext
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 parallel algorithm that require time proportional to $\log^2 n$. So parallel bubble sort 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 the elements 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:

\[
\begin{array}{ccccccccccc}
\text{bucket} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
221 & 223 & & & 148 & & & \\
071 & & & & & 138 & & \\
131 & & & & & & &
\end{array}
\]

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

Now, the digits in the second position are processed:

\[
\begin{array}{ccccccccccc}
\text{bucket} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
221 & 131 & 148 & & & & & \\
071 & & & & & & &
\end{array}
\]

Again, the buckets are catenated: 221,223,131,138, 148,071.
Then, the digits in the third position are processed:

\[
\begin{array}{ccccccccccc}
\text{bucket} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
071 & 131 & 221 \\
138 & 223 \\
148
\end{array}
\]

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 \texttt{pack} intrinsic function. \texttt{Pack(array, mask)} returns a one-dimensional array containing the elements of \texttt{array} to pass the \texttt{mask}.

Thus, assuming that the sequence to be sorted is in vector \texttt{a}, and that the elements are in base \texttt{b} and contain \texttt{d} digits each, we can proceed as follows:
do i=1,d
  m_0(1:n) = \textit{the digit in a(1:n) with weight }b^{i-1}\textit{ is }0
  m_1(1:n) = \textit{the digit in a(1:n) with weight }b^{i-1}\textit{ is }1
  \ldots
  m_{b-1}(1:n) = \textit{the digit in a(1:n) with weight }b^{i-1}\textit{ is }b-1
  a= (/pack(a,m_0), pack(a,m_1), \ldots, pack(a,m_{b-1}) /)
end do

\textbf{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 pointers. 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 containing 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 follows¹:

```plaintext
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:

```plaintext
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 take 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
\[
\begin{align*}
do & \ i=1, n \\
do & \quad j=1, n \\
& \quad \quad a(i,j)=a(i-1,j)+1 \\
& \quad \quad \text{end do} \\
& \text{end do}
\end{align*}
\]
can be represented by the following graph:
From the graph it is clear that for each $i$ there is a vector operation in $j$.

\begin{verbatim}
do i=1,n
   a(i,1:n)=a(i-1,1:n)+1
end do
\end{verbatim}

A second example:

\begin{verbatim}
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
\end{verbatim}
Now for each \(j\) there is a vector operation:

\[
\text{do } j=1,n \\
\quad a(1:n,j) = a(1:n,j-1) + a(2:n+1,j+1) + b(i) + c(1:n) \\
\text{end do}
\]

A more complicated case:

\[
\text{do } i=2,n \\
\quad \text{do } j=2,n \\
\quad\quad a(i,j) = a(i,j-1) + a(i-1,j) \\
\quad \text{end do} \\
\text{end do}
\]
From the equations:

\[
2 \leq i \leq n \\
2 \leq k - i \leq n \\
k = 4, 5, ..., 2n
\]

We conclude that:

\[
\max(2, k-n) \leq i \leq \min(n, k-2)
\]

From where:

\[
\text{do } k=4, 2*n \\
\text{forall (i=\max(2, k-n):\min(n, k-2)) a(i, k-j)=...} \\
\text{end do}
\]
Another complex example:

\[
\begin{align*}
\text{do } & i=2, n \\
& \quad \text{do } j=2, n \\
& \quad \quad a(i,j) = a(i+1, j-1) + a(i-1, j) + a(i, j-1) \\
& \quad \text{end do} \\
& \text{end do}
\end{align*}
\]
From the equations:

\[ 4 \leq 2i \leq 2n \]
\[ 2 \leq k - 2i \leq n \]
\[ k = 6, 5, \ldots, 3n \]

We conclude that:

\[ \max\left(2, \left\lfloor \frac{k-n}{2} \right\rfloor \right) \leq i \leq \min\left(n, \left\lfloor \frac{k-2}{2} \right\rfloor \right) \]

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