Wavefront Pattern

I. Problem

Data elements are laid out as multidimensional grids representing a logical plane or space. The dependency between the elements, often formulated by dynamic programming, results in computations that resemble a *diagonal sweep* across the elements in the logical plane. The computation starts at a singular point at a corner of the plane and propagates its effect diagonally to other elements. This forms a sweep of computation also known as *wavefront*, and creates nontrivial problem for distribution of work between the parallel processing units.

II. Driving Forces

1. **Workload must be balanced** as the diagonal wavefront computation sweeps the elements.
2. **Processing units** must minimize the idle time while others are executing.
3. **Performance** of the overall system must be efficient.

III. Solution

1. What is Dynamic Programming?

Dynamic programming is a technique to reduce complexity in algorithms that use redundant recursive calls. Instead of calling the same recursive calls again and again, the result of each recursive call is saved and reused when subsequent calls occur. To better illustrate the concept of dynamic programming, we will look at Fibonacci number algorithm.

The Fibonacci number F_n, are defined as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them:

```
FIBONACCI(n)
if (n < 2)
      return n
else
      return FIBONACCI(n-1) + FIBONACCI(n-2)
```

Since one invocation of FIBONACCI will result in two additional series of FIBONACCI calls that recursively invokes other calls, the complexity of the algorithm easily becomes $O(2^n)$. Such exponential complexity is prohibitive for computing for large n, even for parallelized environment.

The complexity of the recursive Fibonacci algorithm can easily be reduced by memorizing the result for each of the calls and reusing them later. This is called

memoization, and is the key idea behind the dynamic programming. Using memoization, we can derive an iterative Fibonacci algorithm with *serial* complexity.

```
FIBONACCI_MEMOIZATION(n)
F(0) = 0
F(1) = 1
for i=2 to n
      F(i) = F(i-1) + F(i-2)
```

In this algorithm, we start from the base case bottom-up, rather than starting from *n* and dividing it into recursive calls. The base case is *memorized* into F and reused in computation of other elements. Computed elements are stored and reused for the subsequent computation steps as shown in Figure 1.

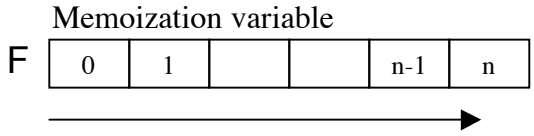

Figure 1. Iterative Process for Fibonacci Number Computation

2. Wavefront

The same concept can be applied to family of dynamic programming involving more than one dimensional memoization variable. Now the memoization involves saving the results for recursive calls involving two parameters. This *often* creates a pattern of computation advancing diagonally on the multidimensional memoization variable space as shown in figure 2. We call this diagonal sweep wavefront.

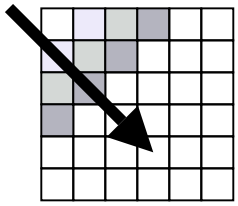

Figure 2. Diagonal computational sweep--Wavefront.

3. Data Distribution

For parallel formulation of wavefront computation, work distribution becomes a critical factor. Work distribution must minimize the idle time and ensure equal amount of work has been assigned to each processing unit. Since the data layout takes the form of multidimensional plane and computation direction is diagonal, computational load at given instance of time differs throughout the sweeping process. For this reason, simple partitions across rows or columns are not encouraged. For example, if simple column partition is used, then the processing

units at the end of the memoization must stand idle until the diagonal wavefront reaches the end column. By the time the wavefront reaches the end column the processing unit that was assigned starting column will sit idle, and so on.

The most widely used data distribution scheme in practice is block cyclic distribution of data. In this scheme, memoization space is partitioned into equal sized column blocks, and distributed cyclically to all the processing units as shown in figure 3. Because of cyclic distribution one processing unit that was assigned less for some partition is compensated by the next partition in the cyclic blocks. For sufficiently small sized blocks, this will create relatively even workload for the processing units.

Processor Distribution

1	2	3	1	2	3
1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10
6	7	8	9	10	11

Figure 3. Cyclic Distribution of wavefront computation. The numbers inside each elements represents time-step that it is computed in.

IV. Related-Pattern

Divide-and-Conquer.

If divide and conquer algorithms are characterized by redundant recursive calls and diagonal dependencies between the calls, use of Wavefront pattern is suggested.

V. Example

The Longest-Common-Subsequence Problem

Given a sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, a subsequence of A can be formed by deleting some entries from A. For example, $\langle a,b,z \rangle$ is a subsequence of $\langle c,a,d,b,r,z \rangle$. The longest common subsequence(LCS) problem can be stated as follows. Given two sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$, find the longest sequence that is a subsequence of both A and B. For example, if $A = \langle c,a,d,b,r,z \rangle$ and $B = \langle a,s,b,z \rangle$, the longest common subsequence of A and B is $\langle a,b,z \rangle$. Finding LCS between two sequences can be a valuable tool in finding valuable information regarding amino acid sequences in biological genes.

Let $F[i, j]$ be the length of the longest common subsequence of the first i elements of A and the first j elements of B. The objective of the LCS problem is to determine $F[n, m]$, where n and m are length of sequences A and B, respectively. Using memoization, we can express the LCS as following:

$$F[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ F[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{F[i, j-1], F[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Given sequences *A* and *B*, consider two pointers pointing to the start of the sequences. If the entries pointed to by the two pointers are identical, then they form components of the longest common subsequence. Therefore, both pointers can be advanced to the next entry of the respective sequences and the length of the longest common subsequence can be incremented by one (2nd case). If the entries are not identical then two situations arise: the longest common subsequence may be obtained from the longest subsequence of *A* and the sequence obtained by advancing the pointer to the next entry of *B*; or the longest subsequence may be obtained from the longest subsequence of *B* and the sequence obtained by advancing the pointer to the next entry of *A*. Since we want to determine the longest subsequence, the maximum of these two must be selected.

The LCS problem can be solved using a two dimensional memoization space *F*. Given by the formulation above, we can conclude that all elements are dependent on :

1) element directly *above* $F[i\text{-}1, j]$
2) element directly *left* $F[i, j\text{-}1]$
3) element directly to the *northwest*. $F[i\text{-}1, j\text{-}1]$

This dependency creates a diagonal computation chain across the problem space, forming a *wavefront*. The dependency and computation direction is shown in figure 4.

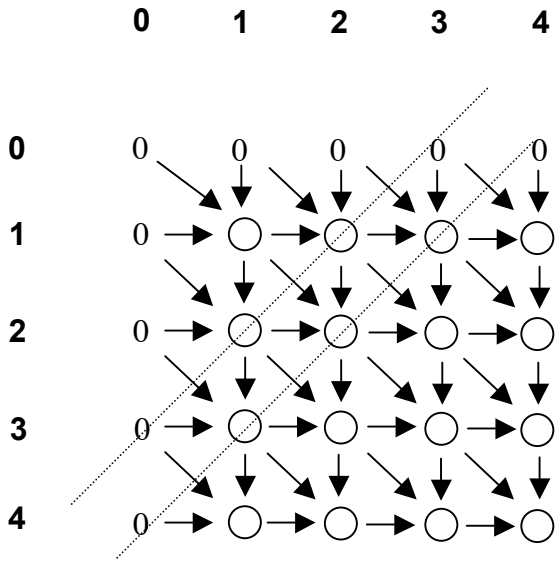

Figure 4. Wavefront formed by dependencies of LCS problem.

The computation starts from $F[0,0]$ and starts filling out the memoization space table diagonally. One example LCS problem can be computed as the following:

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1	1	1	1
W	0	0	0	1	1	1	1	2	2	2	2
H	0	1	1	1	1	1	1	2	3	3	3
E	0	1	2	2	2	2	2	2	3	4	4
A	0	1	2	3	3	3	3	3	3	4	4
E	0	1	2	3	3	3	3	3	3	4	5

Figure 5. Memoization table *F* for computing the LCS of amino acid sequences <H, E, A, G, A, W, G, H, E> and <P, A, W, H, E, A, E>. $F[n, m] = 5$ is the answer to the LCS problem.

The parallel computation for LCS problem will assign each processing unit with a block of columns. Block size will be smaller than m/p, where m is number of columns and p is the number of processing units. Small blocks sizes will allow the columns to be *cyclically* assigned to the processors in round robin fashion. For sufficiently small block sizes, the uneven workload of each columns will be compensated by next cyclic distributed block.