Luke Plewa
CPE 349-01
Dr. Kearns
Algorithm Research

Parallel Needleman-Wunsch Algorithm

Problem Description

The Needleman-Wunsch Aglorithm (1970) solves the problem of aligning strings. The idea is that you can find the optimal alignment to match two strings by finding the best alignment of their substrings. Specifically, the algorithm was created to search for similarities in amino acid sequences. These amino acids compose DNA, and understanding the similarities in DNA helps biologists discover links in genetics. By creating an efficient algorithm to compare strings, Needleman and Wunsch have allowed geneticists a practical way to look at DNA sequences.

There have been improvements on Needleman-Wunsch to further increase the efficiency in aligning. The parallel version of Needleman-Wunsch cuts the sequential process of the algorithm by using parallel computing. DNA sequences are long (between 40KB to 60KB) and sequential computation on them is expensive. Let N be the length of the first string and M be the length of the second. The parallel algorithm increases the efficiency of the Needleman-Wunsch algorithm from O(NxM) to O(N+M), but uses the same amount of memory space as the original, Θ(NxM) (Naveed, Siddiqui, & Ahmed).

Algorithm Explanation

The Needleman-Wunsch algorithm uses the dynamic programming design paradigm to create a solution. This is the recurrence relation used to compute alignment score:

$$C(i, j) = \max \{ C(i-1, j-1) + S(i, j), \quad \text{[diagonal]}$$
$$C(i-1, j) + g, \quad \text{[up]}$$
$$C(i, j-1) + g \} \quad \text{[left]}$$

This recurrence relation builds the score matrix used by the algorithm. G is the gap penalty and S is the mismatch penalty. These are predetermined based on the needs of the user. Changing these variables will affect the decision of the optimal alignment. This approach is bottom-up, using the best alignment of subsequences to find the optimal alignment of the entire sequence. In the Competitors and Analysis section, this recurrence relation will be compared to the recurrence relation of edit distance.

The parallel algorithm uses 5 CPUs to reduce the sequential computation required to achieve this dynamic programming implementation. One contains global memory and the other four calculate. The initializations of the matrices are performed in parallel with four CPUs. The values are then computed and entered in parallel and the division is shown in figure 2.

2,2							
3,2	2,3						
4,2	3,3	2,4					
5,2	4,3	4,3	2,5				
6,2	5,3	4,4	3,5	2,6			
7,2	6,3	5,4	4,5	3,6	2,7		
8,2	7,3	6,4	5,5	4,6	3,7	2,8	
9,2	8,3	7,4	6,5	5,6	4,7	3,8	2,9
9,3	8,4	7,5	6,6	5,7	4,8	3,9	
9,4	8,5	7,6	6,7	5,8	4,9		
9,5	8,6	7,7	6,8	5,9			
9,6	8,7	7,8	6,9				
9,7	8,8	7,9					
9,8	8,9						
9,9							

Fig. 2: Parallel Needleman-Wunsch (Bahria University Islamabad, Pakistan)

With parallel computing, the computations of the matrices are performed in multiple threads.

Figure 2 displays the levels of these threads as they are closed to compute the entries in the score matrix. The last thread to close is the one open for (9, 9) because it relies on (9,8), (8,9), and (8,8) to close. The longest sequence on this graph is the central thread because of its increased amount of neighbor threads.The length of the chart shows that the algorithm can be more efficient with the use of more CPUs. This difference in sequence length creates the improvement in time efficiency from the original O(nm) to O(n+m).

Pseudocode

```
//parallel computation of optimal global alignment
//input: the two strings to align length n and m
//output: the optimal alignment
PNW(seq1, seq2)
// for loops handled in parallel across 4 CPUs
For i=2 to n
        DataArray [0,i] = Seq1[i-2]
For j=2 to m
        DataArray [j,0] = Seq1[i-2]

For i=2 to n
        PointerArray [0,i] = Seq1[i-2]
For j=2 to m
        PointerArray [j,0] = Seq1[i-2]

DataArray [1,1] = 0
Temp = 0

For i=2 to n
        Temp = Temp + g
        DataArray [1,i] = Temp
Temp = 0
For j=2 to m
        Temp = Temp + g
        DataArray [j,1] = Temp
PointerArray [1,1] = 0
Temp = 0
For i=2 to n
        Temp = Temp + g
        PointerArray [1,i] = Temp
Temp = 0
For j=2 to m
        Temp = Temp + g
        PointerArray [j,1] = Temp

//builds the pointer array, this task is assigned and divided between CPUs
int max(int i, int j)
        Diagonal = DataArray [i-1,j-1]
        Up = DataArray [i-1,j]
        Left = DataArray [i,j-1]
        Diagonal = Diagonal+  s(i, j)
        Up = Up + g
        Left = Left + g
        If ( Diagonal > Left AND Diagonal > Up )
                PointerArray[i,j]="3"
                return Diagonal
        Else If( Up > Left )
                PointerArray[i,j]="2"
                return Up
        Else
                PointerArray[i,j]="1"
                return Left
```

(Algorithm rewritten from the Department of CS&E, Bahria University Islamabad, Pakistan. The

sequential assignment of parallel calculations is omitted from this section. It checks for the availability of the CPU and assigns it the task of computating the scores as seen in figure 2.)

Design Strategy

The Needleman-Wunsch algorithm follows the **dynamic programming** design. It uses two matrices to find the optimal alignment. The first table is filled with the costs. The second table gets traced to find the optimal alignment starting from the bottom right corner. Figure 1 displays the two tables and the traceback routine used to find the alignment between strings SEND and AND.

After all cells are filled, the score and traceback matrices are:

		S	E	N	D
	0	−10	−20	−30	−40
A	−10	1	−9	−19	−29
N	−20	−9	−1	−3	−13
D	−30	−19	−11	2	3

	S	E	N	D
done	left	left	left	left
up	diag	left	left	left
up	diag	diag	diag	left
up	up	diag	diag	diag

Fig. 1: Dynamic Programming Matrices (Likić 2008)

Competitors and Analysis

Before 1970, the only way to compare amino acid sequences was either tediously by eye-matching or using inefficient **brute-force** algorithms. The comparison in efficiency is O(nm) of dynamic programming to exponential costs of brute-force. This revolutionized bioinformatics by finding a more efficient solution. The parallel algorithm has the best time efficiency at O(n+m) compared to the following approaches.

The original algorithm has been adapted to fit specific cases in bioinformatics. **Smith-Waterman** is a modification that looks for **local alignments**. Instead of taking three variables in the recurrence relation, Smith-Waterman uses the maximum of four variables. The fourth variable is the integer zero. This allows the comparison to occur at any point in the strings, it does not have to touch any of the edges.

A similar algorithm is **Hirschberg's algorithm** which solves sequence alignment using a **divide and conquer** technique. It has equal time efficiency to the original Needleman-Wunsch, O(nm), but has more efficient memory use. Needleman-Wunsch uses a 2-D array of Θ(nm) memory space, but Hirschberg's uses a single array Θ(min(n,m)) for memory space.

Hirschberg's uses **Levenshtein edit distance** to compute the differences in sequences. The use of edit distance is a similar algorithm to Needleman-Wunsch for string comparison that also depends on dynamic programming design. The recurrence relations are nearly identical except edit distance looks for the minimum costs of substitutes instead of the maximum score of alignment.

Works Cited

Likić, V. (2008). The Needleman-Wunsch algorithm for sequence alignment. Lecture given at the 7th Melbourne Bioinformatics Course, Bio21. Molecular Science and Biotechnology Institute, University of Melbourne.

Naveed, T., Siddiqui I., Ahmed, S., Parallel Needleman-Wunsch Algorithm for Grid. Department of Computer Sciences & Engerineering, Bahria University Islamabad, Pakistan. Retrieved from: http://www.gridbus.org/~alchemi/files/Parallel%20Needleman%20Algo.pdf.

Needleman, S. and Wunsch, C. (1970). "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48(3):443-53