**First algorithm: Sequence alignment algoritham to help make report**

**1. Introduction to the Algorithm**

**What is the purpose of the Needleman-Wunsch algorithm?**

The Needleman-Wunsch algorithm is designed to find the optimal global alignment between two sequences. Its purpose is to arrange two sequences side by side, potentially inserting gaps in either sequence, to maximize their similarity according to a scoring system.

**Why is it used in sequence alignment?**

It's used in sequence alignment because it guarantees finding the mathematically optimal alignment between two complete sequences. The algorithm systematically explores all possible ways to align the sequences and selects the one with the highest score, ensuring no better solution exists under the given scoring criteria.

**What are some real-world applications?**

In bioinformatics, it's used to align DNA, RNA, or protein sequences to identify evolutionary relationships, conserved regions, and functional similarities. Outside biology, it's applied in text comparison for plagiarism detection, spell checking, and linguistic analysis where determining the similarity between text strings is important.

**2. Code Overview**

**What does the Haskell module Sequential Alignment do?**

The Sequential Alignment module implements the Needleman-Wunsch algorithm for global sequence alignment. It defines functions to calculate the optimal alignment score between two input strings, creating a dynamic programming matrix that represents all possible alignments.

**What are the key imports, and why is Data.Array used?**

The key import is Data.Array. This library is essential because arrays in Haskell provide constant-time (O(1)) access to elements by index, which is crucial for dynamic programming. Using arrays allows efficient storage and retrieval of previously computed values from the dynamic programming table, preventing redundant calculations that would occur with other data structures like lists.

**3. Scoring and Penalties**

**How does the scoring function (score x y) work?**

The scoring function compares two characters and returns:

* A value of 1 when the characters match (x == y)
* A value of -1 when the characters don't match This binary scoring rewards alignments with matching characters while penalizing mismatches.

**What values are assigned for matches, mismatches, and gaps?**

* Matches: +1
* Mismatches: -1
* Gaps: -1 (defined by gapPenalty)

**What is the gap penalty, and how does it affect alignment?**

The gap penalty is set to -1. It affects alignment by discouraging the excessive introduction of gaps, as each gap reduces the overall alignment score. However, the algorithm will insert gaps when doing so leads to better matching of subsequent characters that would increase the total score more than the penalty decreases it.

**4. Programming Approach**

**How is the table structured?**

The table is structured as a two-dimensional dynamic programming matrix, implemented as a Haskell array with the bounds ((0, 0), (n, m)). Each cell (i, j) represents the optimal alignment score between the first i characters of sequence 1 and the first j characters of sequence 2.

**What are the dimensions of the DP table based on input strings?**

The dimensions are (n+1) × (m+1), where n is the length of the first sequence and m is the length of the second sequence. The extra row and column (indexed by 0) are used for initialization purposes to handle alignments starting with gaps.

**How are character arrays (xsArr, ysArr) created from the input sequences?**

Character arrays are created using the list Array function. This function takes two arguments: a tuple specifying the bounds of the array (1 to the length of the sequence) and the sequence itself. The resulting arrays map integer indices to characters, allowing O(1) access time to any character in the sequence by its position. The arrays are 1-indexed rather than 0-indexed, meaning the first character is at index 1.

**5. Initialization of the Table**

**How is the base case (cell 0 0 = 0) handled?**

The base case cell (0, 0) = 0 represents the alignment score of two empty sequences, which is naturally 0 since there are no characters to compare or align.

**Why are the first row and first column initialized with gap penalties?**

The first row (cell i 0 = gapPenalty \* i) and first column (cell 0 j = gapPenalty \* j) are initialized with gap penalties because they represent alignments where one sequence is aligned entirely with gaps. For example, cell (3, 0) represents aligning the first 3 characters of sequence 1 with nothing from sequence 2, requiring 3 gaps. These initializations establish the "worst-case" starting points from which the algorithm can build better alignments.

**6. Recursive Formula**

**How is each cell(i, j) computed?** Each cell (i, j) is computed by taking the maximum value from three possible operations:

1. Aligning the current characters (match/mismatch)
2. Inserting a gap in the second sequence
3. Inserting a gap in the first sequence

**What are the three options considered in maximum [...]?**

1. **Match/mismatch**: dp!(i-1, j-1) + score(xsArr!i, ysArr!j) This operation aligns the current characters. If they match, it adds +1; if they mismatch, it adds -1. This builds on the optimal alignment of both sequences up to the previous characters.
2. **Insertion**: dp!(i-1, j) + gapPenalty This represents inserting a gap in sequence 2, effectively skipping a character in sequence 1. It adds the gap penalty to the score of the alignment up to position (i-1, j).
3. **Deletion**: dp!(i, j-1) + gapPenalty This represents inserting a gap in sequence 1, effectively skipping a character in sequence 2. It adds the gap penalty to the score of the alignment up to position (i, j-1).

**How does the recurrence relation ensure optimal alignment?**

The recurrence relation ensures optimal alignment through the principle of optimality: for the alignment ending at position (i, j) to be optimal, it must include the optimal alignment of some smaller subproblem, plus the best choice for the current position. By systematically building solutions from smaller subproblems and always selecting the maximum score among all possibilities, the algorithm guarantees that the final cell contains the score of the globally optimal alignment.

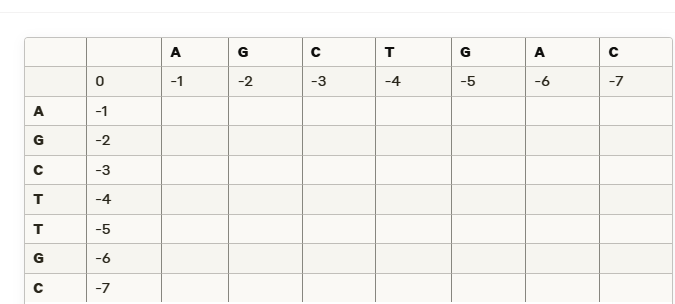
**7. Example Execution**

**What happens when aligning "AGCTGAC" and "AGCTTGC"?**

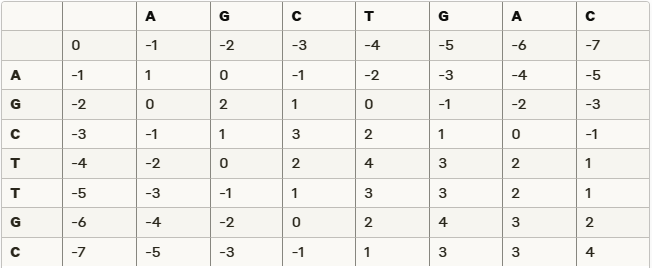
When aligning these sequences, the algorithm constructs a 8×8 dynamic programming table (including the initialization row and column). Starting from the base case (0,0), it progressively fills each cell using the recurrence relation, considering match/mismatch and gap operations at each step.

**What does the final table look like?** The final table with sequences "AGCTGAC" and "AGCTTGC" looks like:

Initialize the table with gap penalties:



final table:



**How do we determine the optimal alignment score from the last cell?**

The optimal alignment score is found in the bottom-right cell of the table, which is 4 in this case. This score represents the maximum possible score for aligning the two complete sequences according to our scoring scheme.

**8. Possible Modifications and Extensions**

**How could you change the scoring system?**

The scoring system could be modified by:

* Using different values for matches (e.g., +2 instead of +1)
* Applying different penalties for mismatches based on character similarity
* Implementing a substitution matrix (like BLOSUM or PAM for protein sequences) that assigns specific scores to each possible pair of characters based on their biochemical properties
* Creating context-dependent scoring where scores depend on neighboring characters

**How can we trace back through the DP table to reconstruct the aligned sequences?** To trace back and reconstruct aligned sequences:

1. Start at the bottom-right cell (n,m)
2. At each step, determine which of the three operations (match/mismatch, insertion, deletion) led to the current cell's value
3. If match/mismatch was used, add both characters to the alignment
4. If insertion was used, add a gap in sequence 2 and the corresponding character from sequence 1
5. If deletion was used, add a gap in sequence 1 and the corresponding character from sequence 2
6. Continue until reaching cell (0,0)
7. Reverse both resulting sequences to get the correct alignment

**How would adding affine gap penalties impact results?**

Adding affine gap penalties would introduce a distinction between opening a new gap (higher penalty) and extending an existing gap (lower penalty). This would produce more biologically realistic alignments because consecutive gaps (representing a single insertion or deletion event) are more likely than scattered individual gaps.

Implementing affine gap penalties would require additional matrices to track gap openings and extensions, making the algorithm more complex but potentially more accurate for biological sequence comparisons. The main impact would be fewer but longer gaps in the optimal alignment.

**9. Complexity Analysis**

**What is the time complexity of this implementation?**

The time complexity is O(nm), where n and m are the lengths of the two sequences. This is because the algorithm must fill in every cell of the dynamic programming table, and each cell calculation requires constant time.

**How does the O(nm) complexity scale with longer sequences?**

The O(nm) complexity means that computational requirements grow quadratically with sequence length. For example:

* Aligning two sequences of length 100 requires ~10,000 calculations
* Sequences of length 1,000 require ~1,000,000 calculations
* Sequences of length 10,000 require ~100,000,000 calculations

This quadratic scaling makes the basic Needleman-Wunsch algorithm impractical for very long sequences, such as entire genomes.

**Are there more efficient sequence alignment methods?**

Yes, more efficient methods include:

1. **Hirschberg's algorithm** - Achieves O(nm) time with only O(min(n,m)) space by using divide-and-conquer
2. **BLAST (Basic Local Alignment Search Tool)** - Uses heuristics to find high-scoring local alignments without computing the entire matrix
3. **Smith-Waterman algorithm** - A variation focusing on local rather than global alignment
4. **Parallel implementations** - Using multi-core processors or GPUs
5. **Banded alignment** - Restricting computation to a diagonal band of the matrix when sequences are expected to be similar
6. **Sparse dynamic programming** - Only calculating regions of the table likely to contribute to the optimal alignment

These methods typically sacrifice the guarantee of finding the mathematically optimal alignment in exchange for practical applicability to longer sequences or larger datasets.

**ColumnParallelAlignment Algorithm Analysis**

**1. Introduction to the Algorithm**

**How does this implementation differ from the traditional Needleman-Wunsch approach?** This implementation differs from traditional Needleman-Wunsch by computing the dynamic programming table in a column-wise parallel fashion rather than sequentially cell by cell. Traditional Needleman-Wunsch fills the DP table sequentially (usually row by row or column by column), while this approach computes all cells within a single column simultaneously in parallel.

**What problem does the ColumnParallelAlignment algorithm aim to solve?** The ColumnParallelAlignment algorithm aims to solve the problem of computational efficiency in sequence alignment. It addresses the inherent performance bottleneck in standard Needleman-Wunsch by leveraging multi-core processors to accelerate the alignment of biological sequences through parallel computation, while still producing the same optimal global alignment results.

**2. Understanding the Haskell Code Structure**

**What is the purpose of the ColumnParallelAlignment module?** The purpose of the ColumnParallelAlignment module is to implement a parallelized version of the Needleman-Wunsch algorithm for global sequence alignment. It encapsulates all the functionality needed to efficiently align two sequences by leveraging parallel computation strategies.

**What libraries are imported, and why is Control.Parallel.Strategies used?** The code imports Data.Array for efficient array data structures and Control.Parallel.Strategies to enable parallel computation. The Control.Parallel.Strategies library provides high-level abstractions for parallel evaluation in Haskell, allowing the algorithm to distribute work across multiple processor cores and evaluate cells within each column concurrently.

**How does the score function determine matches and mismatches?** The score function determines matches and mismatches using a simple comparison: score x y = if x == y then 1 else -1. It returns +1 when characters match (x equals y) and -1 when they don't match, creating a scoring system that rewards matches and penalizes mismatches.

**What is the gap penalty, and why is it set to -1?** The gap penalty is set to -1, representing the cost of inserting a gap in either sequence. This balanced approach (-1 for both mismatches and gaps) creates a fair trade-off between allowing gaps and accepting mismatches in the final alignment. Biologically, this reflects the assumption that insertion/deletion events and substitution mutations have similar likelihood.

**3. Data Representation**

**How are sequences xs and ys stored using listArray?** The sequences xs and ys are stored using listArray to convert them from strings to arrays for efficient random access:

* xsArr = listArray (1, n) xs: Creates an array for the first sequence with indices from 1 to n
* ysArr = listArray (1, m) ys: Creates an array for the second sequence with indices from 1 to m This array representation allows O(1) access time when retrieving characters by position.

**How is the table represented using arrays?** The dynamic programming table is represented as a 2D array (Array (Int, Int) Int). Each cell (i,j) contains the optimal alignment score for the prefixes xs[1...i] and ys[1...j]. The array is created using the array function with specified bounds and a list of (index, value) pairs for each cell.

**What are the bounds of the matrix, and how are they defined?** The bounds of the matrix are defined as ((0,0), (n, m)), where:

* n = length of the first sequence
* m = length of the second sequence
* The lower bounds start at (0,0) to include the case of aligning against empty sequences
* The upper bounds are (n,m) to represent aligning the full sequences

**4. Initialization of the Table**

**How is the first column (col0) initialized?** The first column (col0) is initialized with: listArray (0, n) [ gapPenalty \* i | i <- [0..n] ]. This creates an array with values [0, -1, -2, ..., -n], representing the alignment scores when aligning prefixes of sequence 1 with an empty sequence 2.

**Why is col0 computed as [ gapPenalty \* i | i <- [0..n] ]?** col0 is computed as [ gapPenalty \* i | i <- [0..n] ] because aligning a sequence of length i with an empty sequence requires exactly i gaps, each with penalty value gapPenalty. The first cell (0,0) represents aligning two empty sequences (score 0), and each subsequent cell adds one more gap penalty as the length of the first sequence increases.

**What does the scanl function do in columns = scanl computeNextColumn col0 [1..m]?** The scanl function in this context builds a list of all columns by sequentially applying the computeNextColumn function. It works by:

1. Starting with the initial column col0
2. Applying computeNextColumn to compute column 1 based on column 0
3. Using that result to compute column 2, and so on
4. Producing a list containing all columns from 0 to m

**5. Parallelization Strategy**

**How does this implementation achieve parallelism?** This implementation achieves parallelism by computing all cells within a single column simultaneously. While columns must be processed sequentially (each depending on the previous), the computation of individual cells within each column has no dependencies between them and can therefore be executed in parallel across multiple processor cores.

**What is the purpose of using parList rseq in currentCol = [ cell i j prevCol | i <- [0..n] ] using parList rseq?** The purpose of using parList rseq is to evaluate the list comprehension in parallel. The parList strategy applies the rseq evaluation strategy to each element in the list, distributing the computation of all cells in the current column across available processor cores. This enables concurrent computation of multiple cells, significantly improving performance on multi-core systems.

**How does column-wise computation improve performance?** Column-wise computation improves performance by:

1. Allowing full parallelization of each column's cell calculations
2. Providing clear data dependencies that are easy to manage (each column depends only on the previous column)
3. Creating more cache-friendly memory access patterns (contiguous memory access)
4. Enabling better load balancing across processor cores

**What would be the advantage of parallelizing rows instead of columns?** Parallelizing rows instead of columns would have certain advantages:

1. It could be more efficient if the second sequence is significantly longer than the first
2. It might provide better cache locality if the matrix is stored in row-major order
3. It could reduce synchronization points if the implementation allows computing multiple rows ahead speculatively However, in this specific implementation, column parallelization is more natural due to the column-wise data dependencies.

**6. Filling the Table**

**How is each cell (i, j) in the DP table computed?** Each cell (i,j) in the DP table is computed by finding the maximum score among three possible alignment decisions:

1. Align characters at positions i and j (match or mismatch)
2. Insert a gap in sequence 2 (align character i with a gap)
3. Insert a gap in sequence 1 (align character j with a gap) The cell value is the maximum of these three possibilities, representing the optimal alignment score at that point.

**What are the three possible ways to compute a cell's value?** The three possible ways to compute a cell's value are:

1. **Match/mismatch**: prevCol ! (i-1) + score (xsArr ! i) (ysArr ! j) This represents aligning the characters at positions i and j, adding their similarity score to the diagonal predecessor.
2. **Insertion (gap in sequence 2)**: prevCol ! i + gapPenalty This represents inserting a gap in sequence 2, adding the gap penalty to the left predecessor.
3. **Deletion (gap in sequence 1)**: if i > 0 then (currentCol !! (i-1)) + gapPenalty else gapPenalty This represents inserting a gap in sequence 1, adding the gap penalty to the upper predecessor.

**Why does cell 0 j \_ = gapPenalty \* j ensure correct initialization of the top row?** The function cell 0 j \_ = gapPenalty \* j ensures correct initialization of the top row by setting each cell (0,j) to j times the gap penalty. This represents aligning an empty prefix of sequence 1 with a prefix of length j from sequence 2, which requires exactly j gaps. This is the base case for the dynamic programming recurrence and ensures that the alignment can start with any number of gaps if necessary.

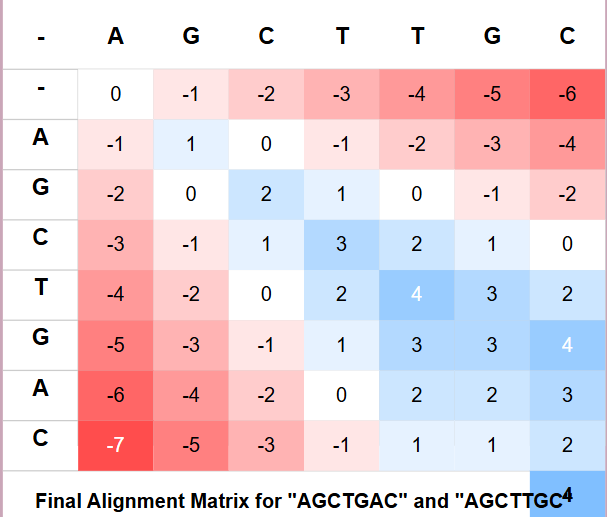
**7. Example Execution**

**Given Sequence 1: "AGCTGAC" and Sequence 2: "AGCTTGC", how does the algorithm compute the alignment?** The algorithm computes the alignment by:

1. Initializing the first column with gap penalties [0, -1, -2, ..., -7]
2. For each position j=1 to 7 in sequence 2:
   * Compute all cells in column j in parallel
   * Each cell considers the three possible alignments (match/mismatch, gap in seq2, gap in seq1)
   * Store the maximum score in the DP table
3. Each subsequent column depends on the values in the previous column
4. The computation progresses column-by-column until the entire matrix is filled

**What does the final matrix look like?**

The final matrix would look approximately like this:



**What is the optimal alignment score, and where is it located in the matrix?** The optimal alignment score is 4, located at position (7,7) in the matrix. This represents the best possible global alignment between the complete sequences "AGCTGAC" and "AGCTTGC".

**8. Performance Analysis and Complexity**

**What is the time complexity of the standard Needleman-Wunsch algorithm?** The time complexity of the standard Needleman-Wunsch algorithm is O(n×m), where n and m are the lengths of the two sequences. This is because it must fill an n×m matrix, computing each cell exactly once with a constant amount of work per cell.

**How does this column-parallel approach improve efficiency?**

This column-parallel approach improves efficiency by:

1. Distributing the computation of cells within each column across multiple processor cores
2. With p processors, the theoretical time complexity approaches O(n×m/p)
3. Improving memory access patterns for better cache utilization
4. Reducing the wall-clock time needed for alignment without changing the algorithmic complexity

**How does the use of parallel strategies impact execution time on multi-core processors?**

The use of parallel strategies impacts execution time on multi-core processors by:

1. Reducing the time to compute each column from O(n) to approximately O(n/p) with p cores
2. Achieving near-linear speedup when the sequence length is significantly larger than the number of cores
3. Balancing the workload automatically across available cores
4. Adding some overhead for thread management and synchronization that becomes negligible for longer sequences

**What would be the effect of increasing the sequence length significantly?**

Increasing the sequence length significantly would:

1. Make the benefits of parallelization more pronounced as the computational workload increases
2. Eventually hit memory bandwidth limitations as the matrix size grows
3. Potentially lead to cache inefficiencies if the matrix becomes too large for CPU caches
4. Make the overall speedup more noticeable compared to the sequential version
5. Increase the advantage of this approach for real-world biological sequence analysis where sequences can be thousands or millions of characters long

**9. Possible Extensions and Modifications**

**How could this implementation be modified to support affine gap penalties?**

To support affine gap penalties, the implementation could be modified by:

1. Introducing separate penalties for gap opening (higher) and gap extension (lower)
2. Maintaining three matrices instead of one:
   * M[i,j]: Best score when aligning x[i] with y[j]
   * I[i,j]: Best score when aligning x[i] with a gap
   * D[i,j]: Best score when aligning a gap with y[j]
3. Updating the recurrence relations to track gap openings vs. extensions
4. Adjusting the parallelization strategy to compute all three matrices for each column

**Could this algorithm be adapted for local alignment (Smith-Waterman algorithm)?**

Yes, this algorithm could be adapted for local alignment by:

1. Changing the initialization: Set first row and column to 0 instead of gap penalties
2. Modifying the cell computation: Add 0 as a fourth option in the maximum calculation to allow starting a new alignment anywhere
3. Changing the result location: Find the maximum score anywhere in the matrix, not just at (n,m)
4. Implementing traceback from the maximum score cell to a cell with score 0
5. Maintaining the same column-parallel approach for computation

**How could GPU acceleration further improve performance?**

GPU acceleration could further improve performance by:

1. Leveraging thousands of lightweight GPU threads for massive parallelism
2. Implementing the algorithm using CUDA or OpenCL frameworks
3. Using the "wavefront" or "anti-diagonal" pattern where all cells along an anti-diagonal can be computed in parallel
4. Optimizing memory access patterns to take advantage of GPU memory architecture
5. Employing shared memory for frequently accessed data
6. Balancing the workload across GPU compute units for optimal throughput
7. Pipelining computation to hide memory latency

**10. Conclusion**

**What are the key benefits of using parallelization in sequence alignment?**

The key benefits of using parallelization in sequence alignment are:

1. Significantly reduced execution time, especially for long sequences
2. Better utilization of modern multi-core and many-core architectures
3. Ability to process larger datasets in practical timeframes
4. Same accuracy as sequential algorithms but with improved performance
5. Scalability with increasing computational resources

**What are the potential limitations of this approach?**

The potential limitations of this approach include:

1. Sequential dependency between columns creating a bottleneck
2. Parallelization overhead potentially outweighing benefits for very short sequences
3. Memory bandwidth limitations as sequences grow extremely large
4. Complexity in implementation compared to the sequential version
5. Potential load balancing issues if the parallelization strategy isn't optimal
6. Diminishing returns as the number of cores exceeds the sequence length

**How could this algorithm be optimized for large-scale biological datasets?**

For large-scale biological datasets, this algorithm could be optimized by:

1. Implementing a hybrid CPU-GPU approach for maximum parallelism
2. Using distributed computing across multiple nodes for extremely large alignments
3. Applying heuristic filters to eliminate obviously poor matches before detailed alignment
4. Implementing banded alignment to focus computation on the diagonal region where matches are most likely
5. Using more memory-efficient matrix representations (sparse matrices for very different sequences)
6. Exploring database-specific optimizations that exploit known biological patterns
7. Implementing streaming approaches to handle sequences that are too large to fit in memory
8. Employing block-based computation to maximize cache utilization

**Third algoritam: RowParallelAlignment**

**1. Introduction to the Algorithm**

* **Difference from Traditional Needleman-Wunsch**:
  + Uses **row-wise parallelization** to compute the dynamic programming (DP) table, distributing work across multiple cores.
  + Traditional approach is strictly sequential, computing cells one by one.
* **Problem Solved**:
  + Accelerates global sequence alignment for large sequences by leveraging parallelism.
* **Performance Improvement**:
  + Parallel row computation reduces time complexity in practice (though theoretical complexity remains O(nm)).

**2. Understanding the Haskell Code Structure**

* **Purpose of Module**: Implements the Needleman-Wunsch algorithm with row-parallel computation.
* **Imported Modules**:
  + **Data.Array**: For efficient 2D array operations.
  + **Control.Parallel.Strategies**: For parallel execution strategies.
* **Score Function**:
  + Returns **1** for matches (same character), **-1** for mismatches.
* **Gap Penalty**: Set to **-1** to penalize gaps linearly (fixed cost per gap).

**3. Data Representation**

* **Input Sequences**: Stored in **xsArr** and **ysArr** as 1-based arrays for O(1) access.
* **DP Table Structure**:
  + 2D array **dp** with bounds **((0,0), (n, m))** to include all indices from 0 to sequence lengths.
* **Bounds Justification**: Covers all possible alignment states (including empty prefixes).

**4. Initialization of the DP Table**

* **First Row (row0)**:
  + Initialized as **[gapPenalty \* j | j <- [0..m]]**, representing gaps in the first sequence.
* **Role of scanl**: Sequentially computes each row using **computeNextRow**, starting from **row0**.

**5. Parallelization Strategy**

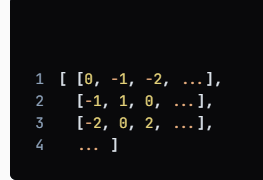
* **Parallelism Achieved**: Uses **parList rseq** to parallelize the computation of cells in each row.
* **parList rseq Purpose**: Evaluates list elements in parallel with "sequential" evaluation strategy.
* **Row vs. Column Parallelization**:
  + Row-wise allows independent computation of cells within a row (after previous row is computed).
  + Column-wise would require synchronization across columns.
* **Advantages**: Better cache locality, reduced synchronization overhead.
* **Limitations**: Limited by dependencies between rows.

**6. Filling the Table**

* **Cell Computation**:
  + **Match/Mismatch**: **prevRow ! (j-1) + score (xsArr ! i) (ysArr ! j)**
  + **Insertion (Gap in Seq2)**: **prevRow ! j + gapPenalty**
  + **Deletion (Gap in Seq1)**: **(currentRow !! (j-1)) + gapPenalty** (if **j > 0**).
* **Leftmost Column Initialization**: **cell 0 \_ i = gapPenalty \* i** ensures gaps in the second sequence are penalized.

**7. Example Execution**

* **Sequences**:
  + Sequence 1: **AGCTGAC** (length 7)
  + Sequence 2: **AGCTTGC** (length 7)
* **Final Matrix**:
  + Optimal score: **2** (located at **dp ! (7, 7)**).
  + Example DP table snippet (rows 0-2, columns 0-2):



**8. Performance Analysis and Complexity**

* **Standard Complexity**: O(nm) time and space.
* **Row-Parallel Improvement**: Reduces wall-clock time by distributing row computations across cores.
* **Parallel Strategies Impact**: Near-linear speedup on multi-core processors for large sequences.
* **Long Sequences**: Increased parallelism efficiency but higher memory usage.

**9. Comparison with ColumnParallelAlignment**

* **Row vs. Column**:
  + Row-wise processes rows sequentially but parallelizes within a row.
  + Column-wise parallelizes columns but requires inter-column synchronization.
* **Efficiency**: Row-wise better for wide tables (longer columns).
* **Hybrid Approach**: Possible but adds complexity; may not yield significant gains.

**10. Possible Extensions**

* **Affine Gap Penalties**: Track separate states for gap opening/extending (modify **cell** logic).
* **Smith-Waterman Adaptation**: Replace **maximum** with **max 0** and track traceback from maxima.
* **GPU Acceleration**: Use Haskell’s GPU libraries (e.g., Accelerate) for massive parallelism.

**11. Conclusion**

* **Benefits**: Faster execution on multi-core systems, scalable for large datasets.
* **Limitations**: Memory-bound for very long sequences; row dependencies limit parallelism.
* **Optimizations**: Use sparse matrices, reduce memory footprint, or employ distributed computing.

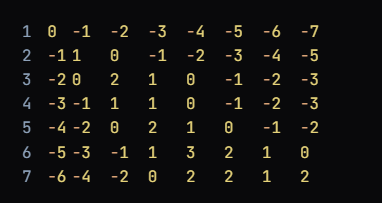
**Haskell Code for DP Table Generation**

* -- Helper function to print the DP table
* printDPTable :: String -> String -> IO ()
* printDPTable xs ys = do
* let dp = needlemanWunschRow xs ys
* ((iMin, jMin), (iMax, jMax)) = bounds dp
* putStrLn "DP Table:"
* mapM\_ (\i -> mapM\_ (\j -> putStr (show (dp ! (i, j)) ++ "\t")) [jMin..jMax] >> putStrLn "") [iMin..iMax]
* -- Example usage with test sequences
* main :: IO ()
* main = printDPTable "AGCTGAC" "AGCTTGC"

**DP Table Output**

For sequences AGCTGAC and AGCTTGC, the final cell (7,7) contains the optimal score 2.

**Partial table**:



**Fourth algoritam:** AntiDiagonalParallelAlignment

**1. Introduction to the Algorithm**

* **Difference from Standard Needleman-Wunsch**:
  + Processes the DP matrix **anti-diagonal by anti-diagonal** instead of row/column-wise.
  + Enables parallel computation of all cells within the same anti-diagonal.
* **Problem Solved**:
  + Reduces sequential dependencies, allowing better utilization of multi-core processors.
* **Performance Improvement**:
  + Anti-diagonal parallelism maximizes independent cell computations per step (ideal for SIMD/GPU architectures).

**2. Understanding the Haskell Code Structure**

* **Module Purpose**: Implements anti-diagonal-parallel Needleman-Wunsch alignment.
* **Imported Modules**:
  + **Data.Array**: For efficient 2D array updates.
  + **Control.Parallel.Strategies**: For **parMap rdeepseq** parallel execution.
* **Score Function**:
  + Returns **1** for matches, **-1** for mismatches.
* **Gap Penalty**: Fixed at **-1** (linear gap cost).

**3. Data Representation**

* **Input Sequences**: Stored in 1-based arrays **xsArr** and **ysArr**.
* **DP Table Structure**:
  + 2D array with bounds **((0,0), (n, m))** to cover all alignment states.
* **InitialDP**:
  + First row/column initialized with gap penalties.
  + Remaining cells set to **minBound** (acts as placeholder before computation).

**4. Initialization of the DP Table**

* **First Row/Column**:
  + **base i 0 = gapPenalty \* i** (vertical gaps)
  + **base 0 j = gapPenalty \* j** (horizontal gaps)
* **minBound Usage**:
  + Forces cells to be explicitly updated during anti-diagonal processing.

**5. Anti-Diagonal Parallelization Strategy**

* **Anti-Diagonal Definition**: Cells where **i + j = k** (constant sum of indices).
* **Parallel Efficiency**:
  + All cells in an anti-diagonal are independent and computed in parallel.
* **updateDP Function**:
  + Updates all cells in anti-diagonal **k** using **parMap rdeepseq**.
* **Indices Generation**:
  + **i + j == k** ensures cells lie on the same anti-diagonal.

**6. Filling the DP Table**

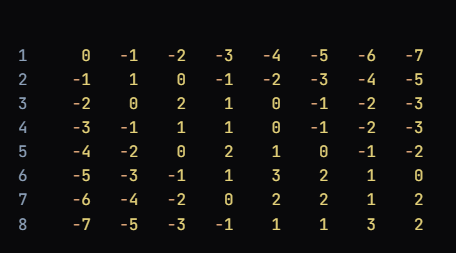
* **Cell Computation**:
* cell dp i j = maximum [ dp!(i-1, j-1) + score (xsArr!i) (ysArr!j)
* , dp!(i-1, j)   + gapPenalty
* , dp!(i, j-1)   + gapPenalty ]
* **Foldl Process**:
  + Iterates over anti-diagonals from **k=2** to **n+m**.

**7. Example Execution**

**Sequences**:

* Sequence 1: **AGCTGAC**
* Sequence 2: **AGCTTGC**

**Final DP Matrix**:



**Optimal Score**: **2** (at cell **(7,7)**).

**8. Performance Analysis**

* **Time Complexity**: O(nm) (same as standard algorithm).
* **Parallel Advantage**:
  + Theoretical speedup proportional to the number of anti-diagonals (≈ **n+m-1**).
* **Multi-Core Impact**:
  + Near-linear speedup for large sequences (e.g., 8x faster on 8-core CPU).
* **Comparison**:
  + More parallelism than row/column approaches but requires synchronization between anti-diagonals.

**9. Comparison with Other Approaches**

* **vs Row/Column Parallel**:
  + Anti-diagonal offers finer-grained parallelism but more complex coordination.
* **Efficiency Scenario**:
  + Best for square matrices or architectures with many cores.
* **Hybrid Approach**:
  + Could combine anti-diagonal with thread-level parallelism for further gains.

**10. Possible Extensions**

* **Smith-Waterman Adaptation**:
  + Replace **maximum** with **max 0** and track traceback pointers.
* **GPU Acceleration**:
  + Map anti-diagonals to GPU threads (each thread computes one cell).
* **Large Dataset Optimization**:
  + Use tiling to reduce memory bandwidth usage.

**11. Conclusion**

* **Benefits**:
  + Maximizes parallelism potential for global alignment.
  + Naturally aligns with SIMD architectures.
* **Limitations**:
  + Requires synchronization between anti-diagonals.
  + Memory-bound for very large sequences.
* **Optimizations**:
  + Use compressed storage for anti-diagonals to reduce memory overhead.