Pairwise sequence alignment with the Smith-Waterman algorithm

Manel Fernández

Intel HPC Software Workshop Series 2016
HPC Code Modernization for Intel® Xeon and Xeon Phi™
February 18th 2016, Barcelona
Pairwise sequence alignment

Comparison applied to *pairs* of biomolecular sequences
- RNA, DNA, or amino-acids

The goal is to identify how similar they are (homology)

Also used on non-biological sequences, such as natural language and financial data
Pairwise comparison of Naegleria NgTet1 vs E. coli AlkB (a–c) and human ABH2 (d–f)

From “Structure of a Naegleria Tet-like dioxygenase in complex with 5-methylcytosine DNA”, by H. Hashimoto Et Al., Nature 506, 391–395 (20 February 2014)
### Pairwise sequence alignment methods

**Global alignment**
- Covers the entire sequences
- Needleman-Wunsch algorithm finds the best global alignment

```
FTFTA L I L L A V A V
F - - T A L - L L A - A V
```

**Local alignment**
- Only covers parts of the sequences
- Smith-Waterman (S-W) algorithm finds the best local(s) alignment(s)

```
FTFTA L I L L - A V A V
- - F T A L - L L A A V - -
```
Smith-Waterman (SW) algorithm

Step 1. Compute the **alignment matrix** or **score matrix** \((\in \mathbb{N})\)

- Best alignment starts on the \([i,j]\) location with the highest score value in the matrix

\[
H[i,0] = H[0,j] = E[0,j] = F[i,0] = 0 \quad 0 \leq i \leq m, 0 \leq j \leq n
\]

\[
H[i,j] = \max \begin{cases} 
E[i,j] = \max \left\{ 
\begin{array}{ll}
0 & \text{Match/Mismatch} \\
H[i-1,j-1] + \sigma(S_1[i], S_2[j]) & \text{Deletion} \quad (S_1[i] \text{ aligned to a gap}) \\
E[i-1,j] - \omega_e & \text{Insertion} \quad (S_2[j] \text{ aligned to a gap}) \\
H[i-1,j] - \omega_o - \omega_e & \text{ } \\
F[i,j-1] - \omega_e & \\
H[i,j-1] - \omega_o - \omega_e & \text{ } \\
F[i,j-1] & \text{ } \\
\end{array} \right. 
\end{cases}
\]

Where

- \(S_1[1..m], S_2[1..n]\) are sequences over alphabet \(\Sigma\)
- \(\sigma(a,b)\) is the similarity function on alphabet letters (substitution matrix)
- \(\omega_o\) and \(\omega_e\) are gap opening and gap extension penalty scheme

Step 2. Find the **optimum local alignment** in the matrix

- From the highest value, go backwards on the direction of used to construct the matrix
- Alignment sequence ends when a matrix cell with zero value is reached
Smith-Waterman algorithm: example

\[ S_1 = \text{AGCACACA}, \ S_2 = \text{ACACACTA} \]

\[ \omega_o = -1, \ \omega_e = 0, \ \sigma(a,b) = \begin{cases} +2, & a = b \\ -1, & a \neq b \end{cases} \]

\[
\begin{array}{c|cccccccc}
--- & --- & --- & --- & --- & --- & --- & --- & --- & --- \\
- & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\
G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\
A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\
C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\
A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\
C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\
A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \\
\end{array}
\]

\[
\begin{array}{c|cccccccc}
--- & --- & --- & --- & --- & --- & --- & --- & --- & --- \\
- & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
G & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
C & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
A & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
C & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
A & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
C & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
A & 0 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\end{array}
\]

**Step 1**

**Step 2**

\[ S_1 = \text{AGCACAC-A} \]

\[ S_2 = \text{A-CACACTA} \]
SW score matrix example

Simple example
- Slightly different sequences
- Similar length (~500 symbols)

Colour gradient for matrix values
- Blue for $H[i,j] = 0$
- Red for high score values

High homology
- About 352 score for this example
- Not necessarily always the case
Other algorithm considerations

Only address Step 1 of the algorithm (*).
- Build the score matrix ("recurrence") and find the highest score ("reduce").
- We will briefly discuss Step 2 later on this presentation.

Algorithm complexity
- Computational complexity: $O(mn)$ (quadratic)
  - "Recurrence" and "reduce" steps can be fused.
- Space complexity: $O(mn)$ (quadratic) ?? $O(m+n)$ (linear)
  - No need to store the whole matrix to find the highest score value.

(*) Code modernization proposed by “Pairwise DNA sequence alignment optimization” by Yongchao Liu and Bertil Schmidt (“High performance parallelism pearls”, Vol. 2, Ch. 4)
Code modernization of S-W algorithm

How to parallelize the score matrix computation?

- Dependency when parallelizing by rows or columns
- **Parallelize over NE-SW diagonals!**

Assuming

- $S_1$ is placed vertically, indexed by $i$, and $\text{length}(S_1) = m$
- $S_2$ is placed horizontally, indexed by $j$, and $\text{length}(S_2) = n$
- $m \leq n$

We can see that

- Diagonal $d = i + j$, where $0 \leq d < m + n - 1$
- Starting row index $srow_d = \max(0, d - n + 1)$
- Ending row index $erow_d = \min(d, m - 1)$
Intra-process (SMP) parallelization

**Scale-and-vectorize approach**

- Partition the matrix into tiles
  - Tile size is $\alpha$ rows $\times$ $\beta$ columns
  - Matrix size is $R = \left\lfloor \frac{m}{\alpha} \right\rfloor$ rows $\times$ $C = \left\lfloor \frac{n}{\beta} \right\rfloor$ columns
- TLP on tiles for every external diagonal
- DLP on every internal diagonal

**Current implementation parameters**

- $C = \#\text{threads}$, $\beta = \left\lfloor \frac{n}{\#\text{threads}} \right\rfloor$
- $\alpha = \text{SIMD}_{\text{length}} \times K$
  - $\text{SIMD}_{\text{length}} = 8$ for AVX2, 16 for AVX-512/IMCI
  - Small $K$ (i.e., $K = 1$) maximize external diagonals with $\#\text{threads}$ tiles, since $\alpha \ll \beta$ and $R \gg C$
S-W in action

Simple example animation
- 8 threads (column tiles)
- Taller tiles than actual ones
Thread parallelism with OpenMP

Private to every thread
GVs are private pointers to global GV[C] arrays

Shared by all threads

C = \#threads horizontal tiles

R = \#vertical tiles

\( C = \#\text{threads horizontal tiles} \)

\( R = \#\text{vertical tiles} \)

\( G = \#\text{vertical tiles} \)

\( H = \#\text{horizontal tiles} \)

\( n-1 \)

\( (d-1) \)

\( (d+1) \)

\( \alpha-1 \)

\( H_{\text{left}} \)

\( F \)

\( G_{\text{Din}} \)

\( G_{\text{Dout}} \)

\( G_{\text{Vin}} \)

\( G_{\text{Vout}} \)
OpenMP implementation

```c
#pragma omp parallel firstprivate(GV,GD) default(shared)
for (d = 0; d < R+C-1; d++)
{
    sExtRow = max(0,d-C+1);
    eExtRow = min(d,R-1);
    Load the per-thread maximum score lmaxScore;
    #pragma omp barrier
    #pragma omp for schedule(static,1) nowait
    for (r = sExtRow; r <= eExtRow; r++)
        Compute all cells in tile (r,d-r);
    Save lmaxScore to a global variable;
    Swap the input and output of GV and GD;
}
Reduce to get the optimal local alignment score;
```

- Threads need to be synchronized before moving to the next external diagonal.
- For every diagonal, every thread will process one tile.
- No need to synchronize at the end since updates are not shared.
- Place together threads with consecutive IDs to improve locality.

S-W within a tile
Auto-vectorization with SIMD

```c
for (int row = 0; row < alpha; row++)
    for (int col = 0; col < beta; col++)
        H[row][col] = max(0, ...);
```

- **loop was not vectorized: vector dependence prevents vectorization**

```c
for (int d = 0; d < (alpha + beta - 1); d++)
    int row = max(0, d - beta + 1);
    int erow = min(d, alpha - 1);
    int col = min(d, beta - 1);
    int ecol = max(0, d - alpha + 1);
    for (; (row <= erow) && (col >= ecol); row++, col--)
        H[row][col] = max(0, ...);
```

- **loop was not vectorized: parallel loop condition operator must be one of <, <=, >, >=, or !=**
Auto-vectorization with SIMD (cont’d)

// main (assume beta > alpha)
for (int d = alpha - 1; d < beta; d++)
   for (int row = 0, int col = d; row < alpha; row++, col--);
   H[row][col] = max(0, ...);

➤ LOOP WAS VECTORIZED, *but*...

Resulting code plenty of **gather/scatter** constructs
  • SIMD elements are not consecutive in memory
  • Scatter instruction does not even exists in AVX2 (software scatter)

What other choices?
  • **Guided** (Cilk Plus)
  • **Low-level vectorization** (SIMD C++ classes, intrinsics)
Vectorization with SIMD intrinsics
Main loop without peel/remainder (IMCI)

Compute column index gCol for current tile;

for (d = alpha - 1; d < beta; d++, gCol++)
{
    // vE[:] = max((GH[gcol].e :: vE[:VL-1]) - ge, (GH[gcol].h :: vH[:VL-1]) - goe)
    vHup = __mm512_alignr_epi32(vH, __mm512_set1_epi32(GH[gCol].h), 15);
    vE = __mm512_alignr_epi32(vE, __mm512_set1_epi32(GH[gCol].e), 15);
    vE = __mm512_max_epi32(__mm512_sub_epi32(vE, vGapE), __mm512_sub_epi32(vHup, vGapOE));

    // vF[:] = max(vF[:] - ge, vH[:]) - goe)
    vF = __mm512_max_epi32(__mm512_sub_epi32(vF, vGapE), __mm512_sub_epi32(vH, vGapOE));

    Compute vScore[:] = \sigma(vS1[:], S2[gCol] :: vS2[0:VL-1]);

    // vH[:] = max(vZero, vD[:]) + vScore[:], vE, vF)
    vH = __mm512_max_epi32(__mm512_max_epi32(vD, vScore), __mm512_max_epi32(vE, vF));
    vD = vHup; // Diagonal values for next iteration

    Save vH[VL-1] and vE[VL-1] on GH (new cell on the last row of the stripe);
}

Save vH[0] and vF[0] on GV.out (new cell on the last column of the stripe);
Inter-process parallelization with MPI

Second level tiling
- Blocks size is $\alpha_{mpi} \times \beta_{mpi}$
- Matrix size is $R_{mpi} = \left[ \frac{m}{\alpha_{mpi}} \right] \times C_{mpi} = \left[ \frac{n}{\beta_{mpi}} \right]$
- $\beta_{mpi} = \left[ \frac{n}{P} \right]$, being $P$ number of MPI ranks

Distributed approach
- Block-level GH, GV, GD distributed buffers
- MPI communication between block diagonals
  - Non-blocking MPI communication primitives
    - MPI_Isend(), MPI_Irecv()
  - MPI_Barrier() for synchronization

Final reduction with MPI_Reduce()
Performance evaluation: preliminary questions

Hint: not all answers need to be correct here...

What’s the ideal target? Xeon, Xeon Phi, or offload?
• Problem is highly parallel, Xeon Phi should be better target
• There should be no big difference in using native or offload approach
  • Computational complexity bigger than space complexity

Compute bound or memory bandwidth bound?
• Kind of “stencil” which tends to be memory bound
• But, proposal does not build a real matrix, so should be compute bound

What’s the expected strong/weak scalability?
• Highly parallel and compute bound might indicate good scalability
• Possible issues for MPI/thread communication/synchronization
Methodology

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Y. Liu and B. Schmidt</th>
<th>Bayncore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Xeon® Processor</td>
<td>E5-2670 (2s, 8c/s, 2ht/c)</td>
<td>E5-2650 v2 (2s, 8c/s, 2ht/c)</td>
</tr>
<tr>
<td>Intel® Xeon Phi™ Coprocessor</td>
<td>4x 5110P (60c, 4t/c, 1GHz)</td>
<td>2x 7120A (61c, 4t/c, 1.2GHz)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux</td>
<td>RHEL Server 6.6 (Santiago)</td>
</tr>
<tr>
<td>Intel® C++ Compiler Version</td>
<td>15.0 ?</td>
<td>15.0.2.164 (build 20150121)</td>
</tr>
<tr>
<td>OpenMP Affinity</td>
<td>Balanced</td>
<td>Balanced</td>
</tr>
<tr>
<td>MPI Library</td>
<td>OpenMPI</td>
<td>Intel® MPI 5.0</td>
</tr>
<tr>
<td>S-W MPI Implementation</td>
<td>Only MPI + Offload + Xeon Phi SIMD source code provided</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI ranks and MPI communication on the processor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Every rank offloads to a coprocessor (1-rank per coprocessor)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \alpha = \text{SIMD}<em>{\text{length}} ), ( \alpha</em>{\text{mpi}} = 128K )</td>
<td></td>
</tr>
<tr>
<td>S-W Input Set (Seq1 vs Seq2) (*)</td>
<td>Seq.</td>
<td>Length</td>
</tr>
<tr>
<td></td>
<td>S4.4</td>
<td>4,411,532</td>
</tr>
<tr>
<td></td>
<td>S4.6</td>
<td>4,641,652</td>
</tr>
</tbody>
</table>

(*) More longer sequences in Y. Liu and B. Schmidt’s performance evaluation
Intra-coprocessor scalability

Smith-Waterman, s4.4 vs s4.6

- Y. Liu and B. Schmidt (Measured)
- Y. Liu and B. Schmidt (Ideal)
- Bayncore (Measured)
- Bayncore (Ideal)

Billion cell updates per second (GCUPS)
Inter-coprocessor scalability

Smith-Waterman, s4.4 vs s4.6

- Y. Liu and B. Schmidt (Measured)
- Bayncore (Measured)
- Y. Liu and B. Schmidt (Ideal)
- Bayncore (Ideal)

Million cell updates per second (GCUPS)

- OpenMP (Xeon)
- OpenMP (Xeon Phi)
- OpenMP + SIMD (Xeon Phi)
- 2x MPI + OpenMP + SIMD (Xeon Phi)
- 4x MPI + OpenMP + SIMD (Xeon Phi)
Inter-coprocessor scalability (cont’d)

Smith-Waterman, MPI strong scaling

- s4.4 vs s4.6
- s23 vs s33
- s23 vs s42
- s23 vs s50
- s33 vs s42
- s33 vs s50
- s42 vs s50
- Average Ideal

Billion cell updates per second (GCUPS)

Number of MPI ranks (i.e., number of Xeon Phi coprocessors)
Compute bound or bandwidth bound?

- Compute bound, but...
  - Also latency bound, since L1-misses not being served by L2
  - Probably from $GH_{in/out}$ on every diagonal
  - Software prefetching might help!
S-W Step 2: Finding optimum local alignment

**Traceback** procedure leading from highest score value
- It would require to store the complete score matrix, which is prohibitive!

Proposal: store *partial* information of score matrix
- Then recompute only the tiles following the alignment path
- Space needed from $m \times n$ down to $C \times R$

\[
\begin{align*}
\alpha - 1 & \quad 0 \quad \beta - 1 \\
\alpha - 1 & \quad 0 \\
\alpha - 1 & \quad 0 \\
m - 1 & \quad \beta - 1 \\
m - 1 & \quad 0 \\
m - 1 & \quad 0 \\
0 & \quad \beta - 1 \\
0 & \quad 0 \\
0 & \quad 0 \\
0 & \quad n - 1 \\
0 & \quad n - 1
\end{align*}
\]
Code modernization: not a single solution
A whole family of parallel programming models

Cache oblivious algorithm
- Cilk Plus, TBB, OpenMP teams with nested parallelism, etc.

```c
void diamond_sw (matrix d)
{
    if (d is small)
        small_sw(d);
    else
    {
        divide d into d0, d1, d2, d3;
        diamond_sw(d0);
        cilk_spawn diamond_sw(d1);
        /*nospawn*/diamond_sw(d2);
        cilk_sync;
        diamond_sw(d3)
    }
}
```
“Future” improvements

Intrinsics are rarely a long-term option

- Compiler **must** vectorize the diagonal-based loop in source code
  - Demand the compiler to vectorize the source code loop (w/ or w/o pragmas)
  - Properly generating “alignr” with the expression: `{ c[1:n] = c[0:n-1]; c[0] = x; }
- File compiler defects at Intel Premier

Observation: values in score matrix are:

- Relatively “small” natural numbers \( n < 2^k, \ n \in \mathbb{N} \land k = \{8,16\} \)
- Stored in 32-bit cells (i.e., 32-bit vector elements)
- Idea: we want longer vectors!

AVX-512 – Byte and Word Instruction (**BWII**) extension

- 512-bit vectors also supporting 16- and 8-bit datatypes (32 and 64 elements!)
  - DNA algorithms will shine on future AVX-512 BWI extension
- Currently supported by Intel compilers, but not by Intel hardware yet
Summary

Intel® Xeon Phi™ speedup
• vs Intel® Xeon Phi™ w/o SIMD: 15.4x
• vs Intel® Xeon® w/o SIMD: 12.1x
• vs “best” Intel® Xeon® w/ SIMD: ?

Benchmark ends to be compute bound
• Not bad intra-process scalability: 60%-70% efficiency
• Excellent inter-process scalability: 80%-90% efficiency

Still room for additional improvements
• Little extra performance by using MPI ranks with native Xeon Phi binaries
• Software prefetching to mitigate current latency problems
• It would be desirable to get rid of intrinsics
  • Using auto-vectorization or Intel® Cilk Plus™ array notation
• Significant future improvement targeting AVX-512 BWI extension