Parallel implementation of the Smith-Waterman algorithm on a multi core architecture

David Livshin
Dalsoft
david.livshin@dalsoft.com

The writing of this article and some of the code design and development it describes took place during the war between Israel and Hamas terrorists. This war follows the vicious terrorist attack on October 7 2023. Being located in Ashkelon, Israel, very close to where the events unfolded, the howl of sirens and explosion of rockets fired on us accompanied my work.

As of this writing, the war still goes on. We will win and wipe Hamas terrorists off the face of the Earth and that will be a win for all the progressive people of the world.

David Livshin

23 October 2023

ABSTRACT

In this article we will analyze the parallel implementations of the Smith-Waterman algorithm for finding similarities in two input strings.

We will establish properties of the Smith-Waterman algorithm with affine gap penalties and relying on that provide a parallel implementation of this algorithm on a multi core architecture utilizing the technology developed while studying stencils (**[5]**).

Algorithm

published on October 23, 2023

See **[1]** for the full description of the Smith-Waterman algorithm. Here we provide the minimal information necessary to describe our work.
Given pair of strings of elements $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$ and given two functions

- $s(a,b)$ - Similarity score of the elements that constituted the two sequences
- $W(k)$ - The penalty of a gap that has length k

construct a scoring matrix H of the size $(n + 1)*(m + 1)$ as following

$H[i,0] = 0$ for $0 \le i \le n$
$H[0,j] = 0$ for $0 \le j \le m$

for $1 \le i \le n$, $1 \le j \le m$
$H[i,j] = \max(M(i,j), LM(i,j), 0)$
were
$UM(i,j) = \max(H[i-k,j] - W(k))$ for $k = 1,...,i-1$
$M(i,j) = \max(H[i-1,j-1] + s(A[i],b[j]), UM(i,j))$
$LM(i,0) = 0$, $LM(i,1) = 0$
$LM(i,j) = \max(H[i,j-l] - W(l))$ for $j > 1$, $l = 1,...,j-1$

While calculating scoring matrix we also store information required for alignment reconstruction (traceback).

Normally an implementation of the Smith-Waterman algorithm accepts as input two sets of sequences SA and SB and for every pair of sequences
$A \in SA$ and $B \in SB$
constructs a corresponding scoring matrix.

We regard the pair of strings of elements $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$ to be processed as two dimensional matrix of the size $(n + 1)*(m + 1)$ with the element (i,j) being:
0 – if i is 0
0 – if j is 0
(a_i,b_j) – if $i > 0$ and $j > 0$

Parallel implementations of the Smith-Waterman algorithm

One of the optimizations of the Smith-Waterman algorithm involves it parallel implementation - see **[2]** for the comprehensive review of this subject.
A number of attempts were made with this respect.

Firstly, by utilizing "parallel" instructions offered by the modern processors e.g. SIMD.

On a multi core processors (with multiple execution threads available) few other techniques can be exploited such as

- distributing each independent pair of sequences to different threads
- observing (see **[3]**) that calculations of $H[i,j]$ and $H[i-1,j+1]$ are independent of each other which means they can be executed in parallel by different threads; this can be extended for more than two points.

In this article we will describe and analyze parallel implementations of the Smith-Waterman algorithm on multi core processors. Firstly, we will provide parallel implementation for an arbitrary penalty of a gap. Then we will provide parallel implementation for a simplified penalty gap (affine gap penalty).

Method

We performed comprehensive study of our parallel implementations of the Smith-Waterman algorithm.
The code is written in **C** and compiled by **gcc** version **12.2** compiler. *OpenMP* was utilized for parallel implementations of our algorithms.

The code was executed under *Linux* operating system running on the 4 cores 8 threads **Intel® Core™ i7-1165G7 @ 2.80GHz** processor. We will refer to this software/hardware configuration as ***reference system***.
It seems that this ***reference system*** reflects what is available to most potential users of this application.

While comparing timing for different kinds of code the following formula was used to calculate improvement of *parallel* implementation over the corresponding *serial* code:

$$1 - (timeOfParallelCode / timeOfReferenceCode) \quad (1)$$

For example if the reference (*serial*) code run for **8.97** seconds and the *parallel* code run for **3.31** seconds the improvement of **63**% was reported. Note that improvements calculated in such a way never exceed **100**%. Also note that improvement of **50**% means that parallel code is twice as fast as the reference code, the improvement of **75**% means that parallel code is four times faster that the reference code.

Comparison was done for identical input sequences and it was attempted to run the code under the same conditions on the system with the minimal possible load.

Generating input data

Although the Smith-Waterman algorithm is used for local sequence alignment of two biological sequences, we approached it implementation as a task of creating parallel code suitable for a multi core processors for the algorithm described in the section **Algorithm**.
For every case in this study the arguments used were integer numbers generated by Dalsoft's Random Testing (*drt*) package (see **[6]**). The input vectors were generated as sequences of random integer numbers in the range ['**a**','**z**'].
The size n of string of elements A is always set to **10000**.
The size m of string of elements B varies and is used as a parameter when evaluating different implementations.

parallel implementation for an arbitrary penalty of a gap

To guide the *parallel* calculation we allocate array of integers $LM\_$ of the size $m+1$ and matrix of integers $UM\_$ of the size $(n + 1)*(m + 1)$. The values stored in these records correspond to the functions UM and LM as following:

$UM\_[i,j] = UM(i,j)$
$LM\_[j] = LM(i,j)$, i being the current row that is processed.

Before starting, initialize $UM\_$ as following

$UM\_[0,j] = 0$ for $0 \le j \le m$

and set $LM\_[0]$, $LM\_[1]$ to **0**.

The calculations are carried out by row; at the row i we calculate

$$H[i,1], H[i,2],...,H[i,m] \quad (2)$$

in this order. Note that calculation of $H[i,j]$ is attempted after calculation of elements

$H[ii,k]$ for $0 < ii < i$ and $0 < k \le m$
$H[i,k]$ for $0 < k < j$

has been completed.
The calculated value of $H[i,j]$ affects $UM\_$ and $LM\_$ as following:

$UM\_[i+m,j] = \max(UM\_[i+m,j], H[i,j] - W(m))$
for $m = 1...n-i$
$LM\_[j+k] = \max(LM\_[j+k], H[i,j] - W(k))$
for $k = 1...n-j$

These calculations don't depend on each other and therefore may be done in parallel. Note that this incremental approach for calculating $UM\_$ and $LM\_$ makes this *parallel* implementation of the Smith-Waterman algorithm to be very different from one described in the section **Algorithm** and not only due to parallel parts of it.

The above suggested way for *parallel* implementation of the Smith-Waterman algorithm in fact is a serial algorithm that performs serial calculation of $H[i,*]$ as specified by (**2**) (with parallel calculation of intermediate values). Therefore for each calculated $H[i,*]$ synchronization is necessary. Synchronization under *OpenMP* is very costly. We attempted to optimize this by creating "pipeline" by calculating more than one $H[i,*]$ before synchronizing threads. This method significantly improves the execution time of our parallel implementation. It has, however, its drawbacks introducing pressure on the underlying memory system. More research is needed to establish the optimal size of the pipeline. Experimenting with the code we discovered that the pipeline size of **40** appears to be optimal for input sizes studied; that is the value that was used while generating runtime results here.

We created *serial* implementation exactly as described in the section **Algorithm**.

Results of optimization

The following table contains comparison between *parallel* and *serial* implementations of the Smith-Waterman algorithm for an arbitrary penalty of a gap.
Each column represents the size m of string of elements B.
Each row represents improvement in % of the *parallel* code over *serial* according to (**1**).

	100	200	300	400	500	600	700	800	900	1000
parallel/serial	80.6	88.47	89	92.6	91.3	91.9	92.1	92.1	92.6	92.7

Needles to say that *parallel* implementation appears to be superior by far than *serial* implementation.

implementation for an affine penalty of a gap

Here we consider the case of the Smith-Waterman algorithm when penalty of a gap function has a special form (an affine penalty of a gap)

$W(k) = u^*k + v,\ u \ge 0,\ v \ge 0$

In **[4]** it is shown that under such an assumption the Smith-Waterman algorithm may be implemented with complexity being $O(m^*n)$ instead of $O(m^{2*}n)$ it normally takes.
Here we will show that the Smith-Waterman algorithm for affine penalty of gap may be efficiently parallelized on a multi core processors.
For all the following algorithms the calculations are carried out by row. Note that calculation of $H[i,j]$ is attempted after calculation of elements

$H[ii,k]$ for $0 < ii < i$ and $0 < k \le m$

has been completed.

First let outline some of the properties of the **max** function we will use:

(1) $\max(a, b) = \max(b, a)$
(2) $\max(a, b, c) = \max(a, \max(b, c))$
(3) $\max(a + v, b + v) = \max(a, b) + v$

The main formula is:

for $1 \le i \le n$, $1 \le j \le m$
$H[i,j] = \max(M(i,j), LM(i,j), 0)$

were

$UM(i,j) = \max(H[i-k,j] - W(k))$ for $k = 1,...,i-1$
$M(i,j) = \max(H[i-1,j-1] + s(A[i],b[j]), UM(i,j))$
$LM(i,j) = \max(H[i,j-l] - W(l))$ for $l = 1,...,j-1$

Denote

$P(i,j) = \max(M(i,j), 0)$

then

$H[i,j] = \max(P(i,j), LM(i,j))$

Note that $P(i,*)$ depends on the values of H for rows $< i$; thus, at the time of processing the i's row these values are known or can be easily calculated; also note that $P(i,j) \ge 0$.

First establish the way UM shall be propagated from a row to the next raw.

$UM[i,j] = \max(H[i-1,j] - W(1), H[i-2,j] - W(2),..., H[1,j] - W(i-1))$

and thus

$UM[i+1,j] = \max(H[i,j] - W(1), H[i-1,j] - W(2),..., H[1,j] - W(i)) =$

Note that

$W(k+1) = W(k) + u$

therefore the above can be written as

$UM[i+1,j] = \max(H[i,j] - W(1), H[i-1,j] - W(1) - u,..., H[1,j] - W(i-1) - u)$

$= \max(H[i,j] - W(1), \max(H[i-1,j] - W(1),..., H[1,j] - W(i-1)) - u)$

And finally

$UM[i+1,j] = \max(H[i,j] - W(1), UM[i,j] - u)$ (3)

Serial implementation for an affine penalty of a gap

Theorem:

for $j > 0$

$LM(i,j+1) = \max(P(i,j) - W(1), LM(i,j) - u)$ (4)

Proof:

By definition

$LM(i,j) = \max(H[i,j-l] - W(l))$ for $l = 1,...,j-1$

so

$LM(i,j+1) = \max(H[i,j+1-l] - W(l))$ for $l = 1,...,j$

or

$LM(i,j+1) = \max(H[i,j] - W(1), H[i,j-1] - W(2), ..., H[i,1] - W(j))$

Note that

$W(k+1) = W(k) + u$

thus the above can be written as

$LM(i,j+1) = \max(H[i,j] - W(1), H[i,j-1] - W(1) - u, ..., H[i,1] - W(j-1) - u) =$

$\max(H[i,j] - W(1), \max(H[i,j-1] - W(1) - u, ..., H[i,1] - W(j-1) - u)) =$

$\max(H[i,j] - W(1), \max(H[i,j-1] - W(1),..., H[i,1] - W(j-1)) - u) =$

$\max(H[i,j] - W(1), LM(i,j) - u))$

As defined above.

$H[i,j] = \max(P(i,j), LM(i,j))$

or

$H[i,j] - W(1) = \max(P(i,j) - W(1), LM(i,j) - W(1))$

Substituting this in the above we get

$LM(i,j+1) = \max(H[i,j] - W(1), LM(i,j) - u))$

$= \max(\max(P(i,j) - W(1), LM(i,j) - W(1)), LM(i,j) - u)$

$= \max(P(i,j) - W(1), LM(i,j) - W(1)), LM(i,j) - u))$

$= \max(P(i,j) - W(1), \max(LM(i,j) - W(1)), LM(i,j) - u))$

)

$W(1) = u + v$ and, because $v \geq 0$, $W(1) \geq u$.

Therefore $LM(i,j) - W(1)) \leq LM(i,j) - u$ and thus

$\max(LM(i,j) - W(1)), LM(i,j) - u)) = LM(i,j) - u$

so

$LM(i,j+1) = \max(P(i,j) - W(1), LM(i,j) - u))$

EndOfProof

The above result is reminiscent of the one that is mentioned in **[4]**. We will use it for serial implementation of the Smith-Waterman algorithm for an affine penalty of a gap as well as a base for for parallel implementation utilizing the technology that was developed in *dpl* - Dalsoft's Parallel Library - see **[5]** for the full description.

Parallel implementation for an affine penalty of a gap

Theorem:

provided that

$W(k) = u^*k + v$, $u >= 0$, $v >= 0$

then for $j > 1$

$LM(i,j) = \max(P(i,j-l) - W(l))$ for $l = 1,...,j-1$ (5)

Proof:

Math induction:

$LM(i,2) = \max(H[i,1] - W(1))$

$= \max(\max(LM(i,1), P(i,1)) - W(1))$

Note that $LM[i,1] = 0$, therefore

$LM(i,2) = \max(\max(0, P(i,1)) - W(1))$

Because $P(i,1) >= 0$ then $\max(0, P(i,1)) = P(i,1)$

and therefore $LM(i,2) = \max(P(i,1) - W(1))$

Assume that

$LM(i,j) = \max(P(i,j-l) - W(l))$ for $l = 1,...,j-1$

according to (4)

$LM(i,j+1) = \max(P(i,j) - W(1), LM(i,j) - u)$

and substituting our assumption:

$LM(i,j+1) = \max(P(i,j) - W(1), P(i,j-1) - W(1) - u,..., P(i,1) - W(j-1) - u)$

note that $W(k) + u = W(k+1)$, therefore

$LM(i,j+1) = \max(P(i,j) - W(1), P(i,j-1) - W(2), ..., P(i,1) - W(j))$

EndOfProof

As mentioned earlier, at the time of processing the ***i***'s row ***P***(***i***,***j***) is known or can be easily calculated. Therefore values involved in calculation of ***LM***(***i***,***j***) according to (**5**) don't depend on each other and calculation of ***LM***(***i***,**2**)…***LM***(***i***,***m***) can be done in parallel. For optimal performance load balancing shall be considered.

Although the above describes parallel implementation of our task, it is hardly optimal. Each ***LM***(***i***,***j***) requires calculation and processing of ***j***-**1** members thus causing the complexity of processing a raw to be O(m^2) and total processing to be O($m^{2*}n$). Therefore the above described parallel implementation does not appear to be useful, has little practical value and we regard it as "a proof of concept" that doesn't merit implementation. In the next section we will describe much more efficient algorithm.

Another parallel implementation for an affine penalty of a gap

According to (**4**)

LM(***i***,***j***+**1**) = **max**(***P***(***i***,***j***) - ***W***(**1**), ***LM***(***i***,***j***) - ***u***)

P(***i***,***j***) depends on the values of ***H*** for rows < ***i***; therefore when processing the raw ***i*** results ***LM***(***i***,*) don't affects it. Thus (**4**) defines a stencil.

Stencils were thoroughly studied in **[5]** (and some of the technology developed there is offered by *dpl* - Dalsoft's Parallel Library). Based on this technology we provide a parallel implementation of the Smith-Waterman algorithm for an affine penalty of a gap.

Results of optimization

We created a *serial* implementation based on (**4**).

The following table contains comparison between *parallel* and *serial* implementations of the Smith-Waterman algorithm for an affine penalty of a gap.
Each column represents the size ***m*** of string of elements ***B***.
Each row represents improvement in % of the *parallel* code over *serial* according to (**1**).

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
parallel/serial	29.2	42.7	49.4	54	56	58	59.5	58	60.3	61

From the above data it is clear that *parallel* implementation is superior to *serial*. The efficiency improves as the size ***m*** of the string of elements ***B*** grows; although, from our experience with parallel code performance, it seems that this growth will come to a halt when the size of the problem will overwhelm the memory handling capacity of ***reference system***.

Now we will establish how well our *parallel* implementation scales by the number of threads available. The following table contains comparison between *parallel* and *serial* implementations of the Smith-Waterman algorithm for an affine penalty of a gap.
The size ***m*** of string of elements ***B*** was fixed to be **5000**.
Each column represents the number of threads utilized when executing parallel code. Each row represents improvement in % of the *parallel* code, executed with the number of threads specified, over *serial* according to (**1**).

	2	3	4	5	6	7	8
parallel/serial	30.3	43.2	52.1	50	51.5	55.2	56

The ***reference system*** we used for this study uses Intel's Hyper-Threading Technology, which is known for not being able to always boost performance.

The above table clearly demonstrates that, when utilizing number of threads that does not exceeds number of cores available (**4** cores available in the ***reference system*** we used), parallel code scales well across threads available. This suggests that on a system with larger number of cores than on our ***reference system*** we will see more spectacular improvements than the one observed here.

REFERENCES

[1] "Smith–Waterman algorithm", [Online]. Available https://en.wikipedia.org/wiki/Smith–Waterman_algorithm

[2] Zeyu Xia, Yingbo Cui, Ang Zhang, Tao Tang, Lin Peng, Chun Huang, Canqun Yang, Xiangke Liao. A Review of Parallel Implementations for the Smith–Waterman Algorithm. Interdisciplinary Sciences: Computational Life Sciences (2022) 14:1–14

[3] Wozniak A (1997) Using video-oriented instructions to speed up sequence comparison. Bioinformatics 13(2):145–150.

[4] O Gotoh, An improved algorithm for matching biological sequences. J MolBiol. 162, 705–708 (1982).

[5] David Livshin. "Dalsoft's Parallel Library", [Online]. Available https://dalsoft.com/dpl.html

[6] David Livshin. "Dalsoft's Random Testing Library", [Online]. Available https://www.dalsoft.com/drt.html