

Parallel Linear Space Algorithm for Large-scale Sequence Alignment

Eric Li¹, Cheng Xu², Tao Wang¹, Li Jin³, Yimin Zhang¹

Intel China Research Center, Intel Corporation, Beijing, China¹

Department of Computer Science, Peking Univ., Beijing, China²

Department of Computer Science, Tsinghua Univ., Beijing, China³

Abstract

Aligning long DNA sequences is a fundamental and common task in modern molecular biology. Though efficient dynamic programming algorithms have been presented to solve this problem, the required space and time still poses a large amount of challenge for large scale sequence alignments. Based on the detailed parallelism analysis of the underlying algorithm, we propose an efficient algorithm, i.e., Parallel Linear Space Alignment (PLSA), to compute the long sequence alignment. The local start point and grid cache can segment the whole sequence alignment problem into several independent subproblems, which dramatically reduces the re-computations of trace back phase and provides more parallelism than the conventional algorithms. Moreover, the global start point helps us to find more than one near-optimal non-intersecting alignments and several parameters can be tuned to achieve better scalability performance over various platforms. Our experiments show that the proposed algorithm can be efficiently parallelized on a PC cluster and exhibit significant execution time saving as well as good speedup performance compared with the existing algorithms.

Keywords:

Sequence alignment; Dynamic programming; Parallel algorithms; Space-efficient; Grid Cache; Start point

1. Introduction

Pair-wise sequence alignment is a fundamental operation in the research of bioinformatics. Sequence can be nucleic acids in a genome or amino acids in a protein which is expressed as a character format, says a string. Pair-wise sequence alignment is capable of identifying the similar and diverged regions between two sequences. From a biological point of view, sequence alignments can be used to probe similar sequences among sequence databases, detect genetic disease, and construct of phylogenetic trees and even more.

Since sequence alignment can be regarded as an equivalence to calculate an edit distance between a pair of strings. Dynamic programming, an efficient algorithm for editing distance problem, is adapted to solve the alignment problem. In 1970 Needleman and Wunsch [13] introduced the first algorithm for the global alignment problem using

dynamic programming. Smith and Waterman [15] adapted this algorithm to the problem of local alignment. As these algorithms based on the idea of dynamic programming have the quadratic complexities with respect to the length of two sequences, they will demand excessive computation power and memory than single computation resource could afford, especially when large sequences such as genome data are used [3, 4, 10]. Though BLAST [14] and FASTA [16] introduce heuristics to speed up the dynamic programming, they lose the alignment sensitivity and are not very suitable for genome alignment due to the huge memory requirement. Some other algorithms are proposed to utilize parallelism to align long sequence without loss of the sensibility of optimal alignment such as Myers [10] does. The general ideas of these algorithms are commonly based on Divide and Conquer, which tries to divide the whole problem into several subproblems and exploit more opportunities of parallelism.

In this paper we propose an efficient algorithm PLSA for long sequence alignment. There are several advantages of our algorithm. First, it can solve the whole problem with linear space by introducing the idea of the global/local start point and grid cache. After full filling the similarity matrix with partial information stored in the grid cache, the whole problem can be partitioned into several independent segments which can be solved in parallel. The algorithm not only reduces the re-computations of trace back period dramatically but also provides more parallelism than FastLSA [1] since all these segments are independent with each other. Moreover, the global start point information can facilitate us to find some near-optimal non-intersecting alignments at the same time. We show that this approach can lead to significant execution time savings and exhibits much better scalability. In addition, we can use a couple of parameters to tune the application's performance over various parallel platforms.

The rest of this paper is organized as follows. In section 2, we introduce the background and related works in parallelization of linear sequence alignment. Section 3 provides the description of the proposed PLSA algorithm. The underlying methods and implementations of parallel PLSA are illustrated in section 4. We evaluate our experimental result and discuss the parameter impact in section 5. Finally, section 6 concludes this work.

2. Background and Related Works

In 1970 Needleman and Wunsch introduced the first algorithm for the global alignment problem using dynamic programming method. Smith and Waterman adapted this algorithm to the problem of local alignment. The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one into another. Consider two strings S_1 and S_2 of length l_1 and l_2 . The smith-waterman algorithm computes the similarity $H(i, j)$ of two sequences which is given by the following occurrences:

$$\begin{aligned}
H(i, j) &= \max \begin{cases} 0 \\ E(i, j) \\ F(i, j) \\ H(i-1, j-1) + sbt(S1_i, S2_j) \end{cases} \\
E(i, j) &= \max \begin{cases} H(i, j-1) - \mathbf{a} \\ E(i, j-1) - \mathbf{b} \end{cases} \\
F(i, j) &= \max \begin{cases} H(i-1, j) - \mathbf{a} \\ F(i-1, j) - \mathbf{b} \end{cases}
\end{aligned} \tag{1}$$

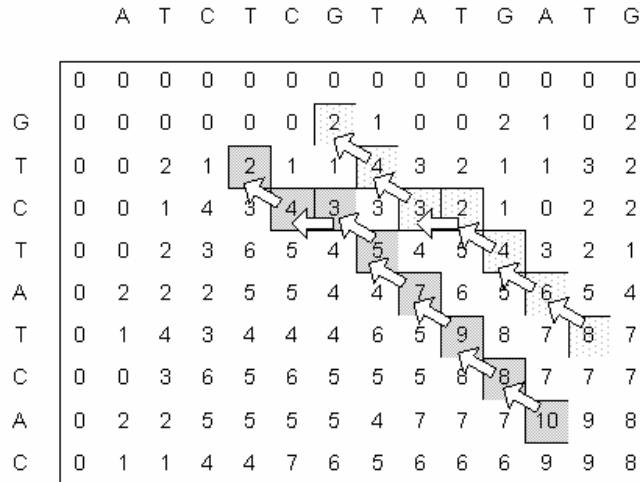
Where $1 \leq i \leq l1, 1 \leq j \leq l2$ and $sbt()$ is substitution matrix of cost values. Initialization of these values is given by:

$$H(i, 0) = E(i, 0) = 0, 0 \leq i \leq l1$$

$$H(0, j) = F(0, j) = 0, 0 \leq j \leq l2$$

Affine gap costs are defined as follows: \mathbf{a} is the cost of the first gap; \mathbf{b} is the cost of the following gaps. Each position of the matrix H is a similarity value. E and F are the cost values of position (i, j) to indicate that it comes from a gap status. E.g., the two sequences of $S1$ and $S2$ generate the similarity matrix and the whole optimal alignment can be traced back as illustrated in Fig 1.

Although able to report all possible alignments between two sequences, these algorithms pose challenges both on computer memory and execution time when handling long sequences such as whole genomes. Take these two sequences as example, the memory complexity for Smith-Waterman algorithm is $O(l1 \cdot l2)$. For aligning sequences of several hundred kilo pair wise sequences would lead to a memory requirement of several Terabytes. However, it is possible to reduce the memory space complexity from quadratic to linear in the length of sequences by using the following algorithms.



Similariy matrix of the alignment of sequence ATCTCGTATGATG and GTCTATCA. k=2, substitution cost of +2 if the two characters are identical and -1 otherwise. Both the first and extension gap penalty are -1. Two traceback paths deliver the non-intersecting optimal and near-optimal local alignments.

Figure 1 Smith-waterman sequence alignment

Hirschberg [6] was the first to use a linear space algorithm to find common substrings. The algorithm uses the basic divide and conquer technique to solve the space problem,

and Myers and Miller [10] applied Hirschberg's linear space algorithm to sequence alignment. Regarding the parallel possibilities in linear space sequence alignment, several algorithms have been proposed such as Martin [9], Arulu [2] and FastLSA etc. Martins *et al.* use equally-sized blocks and the algorithm statically pre-assigns rows of blocks to each processor. It suffers from the similar major drawback as the original full matrix algorithm since the space required is quadratic in the size of sequence. Arulu presents another parallel solution using parallel prefix computations, however it only works well for architectures with fast interconnect communication and has some inevitable load balance problem. FastLSA is basically an extension of Hirschberg's algorithm. It uses some extra space called grid cache to save a few rows and columns of similarity matrix, divides the whole problem into several dependent subproblems and recursively solves each subproblem to get the optimal alignment from the end to the start point. It has less re-computations than Hirschberg's algorithm and the filling similarity matrix computation step can be done in parallel. However, the scalability performance of solving subproblems is not very satisfying since the granularity of recursive subproblems is too fine with more processors. Chen and Schmidt's [5] algorithm builds on the ideas of Arulu and uses the start point of each cell in the similarity matrix in order to determine k near-optimal non-intersecting alignments. After the start and end point of an alignment are determined, they solve the alignment using Hirschberg's algorithm. Therefore its sequential execution time is more than FastLSA and it still has some load imbalance problems with parallel version.

3. Sequential PLSA Algorithm

We propose PLSA algorithm and show how it is different from the traditional algorithms. It uses the grid cache and start point to achieve the minimum sequential execution time and provide more parallelism especially when coping with the trace back phase. In addition, it can output several near-optimal alignment paths after one full matrix filling process. It also introduces several tunable parameters so that the whole algorithm can adapt to different hardware configurations, such as cache size, main memory size and the communicate interconnections.

The basic idea of PLSA is to use more available memory to reduce the re-computation time that need to be done in trace back phase. In particular, it leads to less execution time than FastLSA, which is believed to be the state-of-the-art algorithm for solving linear-space sequence alignment problems.

3.1 Finding near-optimal alignments in linear space

The Smith-Waterman algorithm only computes the optimal local sequence alignment result. However, there are still some other biological important alignments with scores close to the optimal one. For comparison of long DNA sequences the detection of near-optimal non-intersecting local alignments is particularly important and useful. Chen's algorithm uses the start point information to distinguish these different local alignments. However, their recurrences equation is slightly inconvenient since a little more computation and memory have to be used. Therefore we propose a simpler and faster recurrence method. It works as follows:

For each cell (i, j) in the similarity matrix H , define the global start point $Hst(i, j)$ to be the starting position of the local alignment ending at (i, j) . Similar to the Eq (1), the values of $Hst(i, j)$ are calculated by the following recurrence equations:

$$\begin{aligned}
 Est(i, j) &= \begin{cases} Hst(i, j-1) & \text{if } E(i, j) = H(i, j-1) - a \\ Est(i, j-1) & \text{if } E(i, j) = E(i, j-1) - b \end{cases} \\
 Fst(i, j) &= \begin{cases} Hst(i-1, j) & \text{if } F(i, j) = H(i-1, j) - a \\ Fst(i-1, j) & \text{if } F(i, j) = F(i-1, j) - b \end{cases} \\
 Hst(i, j) &= \begin{cases} (i, j) & \text{if } H(i, j) = 0 \\ Hst(i-1, j-1) & \text{if } H(i, j) = H(i-1, j-1) + sbt(S1_i, S2_j) \\ Est(i, j) & \text{if } H(i, j) = E(i, j) \\ Fst(i, j) & \text{if } H(i, j) = F(i, j) \end{cases}
 \end{aligned} \tag{1}$$

In order to determine k near-optimal non-intersecting alignments, the k highest similarity scores together with different start points are stored during the linear-space computation of the similarity matrix. If we get the maximal score at end point (i_{\max}, j_{\max}) , the start point can be easily obtained according to the stored information. Moreover, these k highest scores will not intersect with each other since the Est , Fst and Hst holds different values in the filling similarity matrix period.

Another advantage of the start point is it can solve both the global and local alignment problem whereas Hirschberg and FastLSA can only solve global alignment problems due to lack of the start point information.

3.2 Grid cache and local start point

Since the whole similarity matrix is usually too large to be fully stored in the memory, more time is usually used as a tradeoff to solve the large space problem. In our PLSA algorithm, we use grid cache rather than the whole matrix when computing the large similarity matrix.

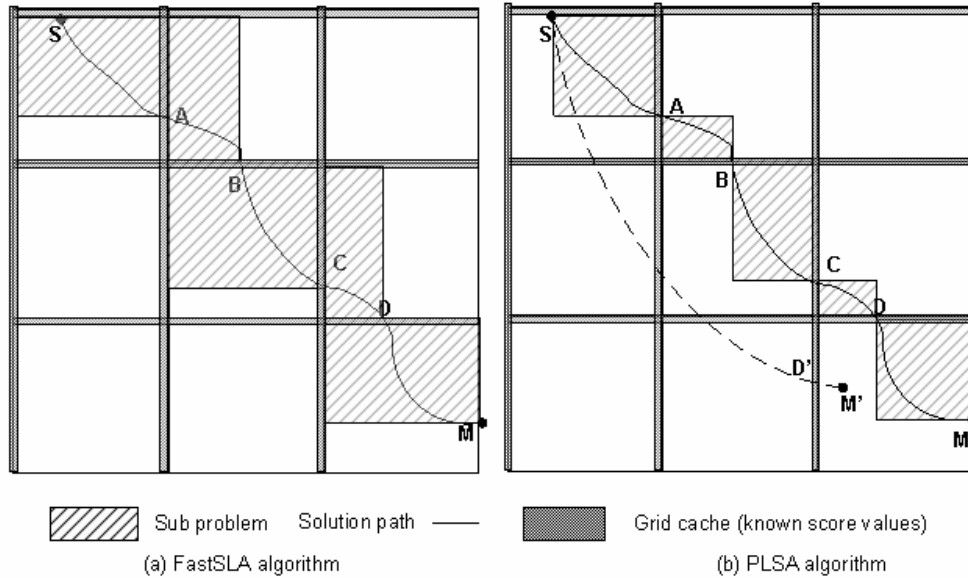


Figure 2 Comparisons between PLSA and FastLSA

The idea of grid cache comes from the FastLSA algorithm. FastLSA uses both column and row bisection method with k divisions on the whole matrix to recursively

divides the matrix into several subproblems. Fig 2(a) shows the case for $k = 3$. The entire similarity matrix is initially computed to save 3 rows and 3 columns as grid cache information during the computation. Once the bottom-right sub-matrix (1/9 of the whole matrix size) is recursively computed and find the optimal path, it will extend from the bottom right region to the top boundary region with the following path $M \rightarrow D \rightarrow C \rightarrow B \rightarrow A \rightarrow S$. However, when D tries to find the backward trace, it does not know where the optimal point locates in its left and up grid. Therefore it has to fill the whole up-left sub-matrix bounded by the grids to find the optimal cell C. The shaded regions indicate the recursive subproblems where re-computations need to be performed. Each of the subproblems, in turn, can be solved recursively using the same procedure. And the solution paths for these subproblems can be concatenated to form a final solution path for the input problem.

In combination with grid cache, we introduce another interesting attribute: local start point, to efficiently reduce the redundant calculations. The idea is very similar to the start point of the optimal alignment. It stores the intersection point of the optimal path and the grids. For distinguishing use, we rename the start point for the optimal alignment as the global start point. For instance, in Fig 2(b), the local start point of M and M' are D and D'. Both of them have the same global start point S. It is obvious that we can not decide whether D or D' is in the optimal path only relying on the global start point information. However, the local start point can distinguish these two cases since they have different local start points respectively.

With the introduction of global and local start points, we can locate the points of the optimal path in the grids and segment the whole problem into several independent subproblems. E.g., in Fig 2(b), when obtaining the maximal score point M, we will search both the last column grid and the last row grid to find its local start point D. Subsequently, we can determine C is the local start point of D in this sub-matrix. Similarly this procedure will be recursively performed to find all the optimal points (D, C, B and A) in the grids. These local start and end points form a series of independent rectangular subproblems continuously, which correspond to subsequence optimal alignment results. Compared Fig 2(a) and Fig 2(b), we can easily observe that the dashed area in PLSA is much smaller than FastLSA, which indicate that fewer re-computations can be achieved in our algorithm. In general, Like FastLSA, our proposed algorithm PLSA also trades space for time. Let $T(m, n, k)$ be the number of matrix entries computed by PLSA when sequence S1 and S2 are aligned using a grid cache with k rows and k columns. The total execution time of PLSA is proportional to $T(m, n, k)$, and the worst case of time $T_w = m \times n \times k / (k - 1)$ since all the subproblems lying on the diagonal will yield the most re-computations. The result is better than FastSLA because its best case can only compete with PLSA's worst one.

To summarize, the combination of grid cache and local/global start point help us to locate the start and end points of several near-optimal non-intersecting alignments after a full computation of the whole similarity matrix, and decompose each optimal path into independent subproblems with correspondent position and status information. In particular, all of these independent subproblems can be worked in parallel, which is a significant advantage over FastLSA.

3.3 Solving Subproblems

In order to make better tradeoff between time and space, we use block as the basic matrix filling unit. A block, something like a 2D matrix, denotes a memory buffer which is available for solving the sequence alignment problem. The width and length of block can be tuned to tailor different memory size, cache size and interconnection configurations. If a problem or subproblem is small enough, we will solve it directly within a block with full matrix filling method. Otherwise it will be decomposed into several descendent subproblems using the method mentioned in section 3.2. These subproblems are very similar to the original problem except that it becomes a global alignment problem other than a local alignment problem since the start and end points are fixed. This will yield some changes on score calculating, initialization, and back tracing procedures. E.g., now the computation of score $H(i, j)$ is given by the following recurrences:

$$\begin{aligned} H(i, j) &= \max \begin{cases} E(i, j) \\ F(i, j) \\ H(i-1, j-1) + sbt(S1_i, S2_j) \end{cases} \\ E(i, j) &= \max \begin{cases} H(i, j-1) - \mathbf{a} \\ E(i, j-1) - \mathbf{b} \end{cases} \\ F(i, j) &= \max \begin{cases} H(i-1, j) - \mathbf{a} \\ F(i-1, j) - \mathbf{b} \end{cases} \end{aligned} \quad (2)$$

All of these continuous subproblems can be solved in parallel since they are independent with each other, and we can subdivide each subproblem until it could be solved within a block size. As the final purpose is to solve the full optimal alignment path, the result of each separated path must concatenate to get the full optimal path.

3.4 Summary of sequential PLSA algorithm

With the combination of global/local start point and grid cache, the proposed algorithm can efficiently solve very large sequence alignment problem. If the matrix size for the input problem is smaller than the block size, it will be solved directly using full matrix filling method. Otherwise, the PLSA algorithm will solve the whole problem and record the global/local start point, score and status information into the grid cache. After that, it splits the problem into several smaller independent subproblems. This procedure will be performed on these subproblems recursively until all these subproblems are solved.

4. Parallel PLSA algorithm

To further improve alignment performance, our sequential PLSA algorithm can be efficiently parallelized. The mapping of large scale sequence alignment to the paralleled architecture consists of two parts: forward calculating the whole similarity matrix and backward solving subproblems to find trace path phase. Fig.3 illustrates the flowchart of the parallel implementation of our algorithm for pair-wise sequence alignment.

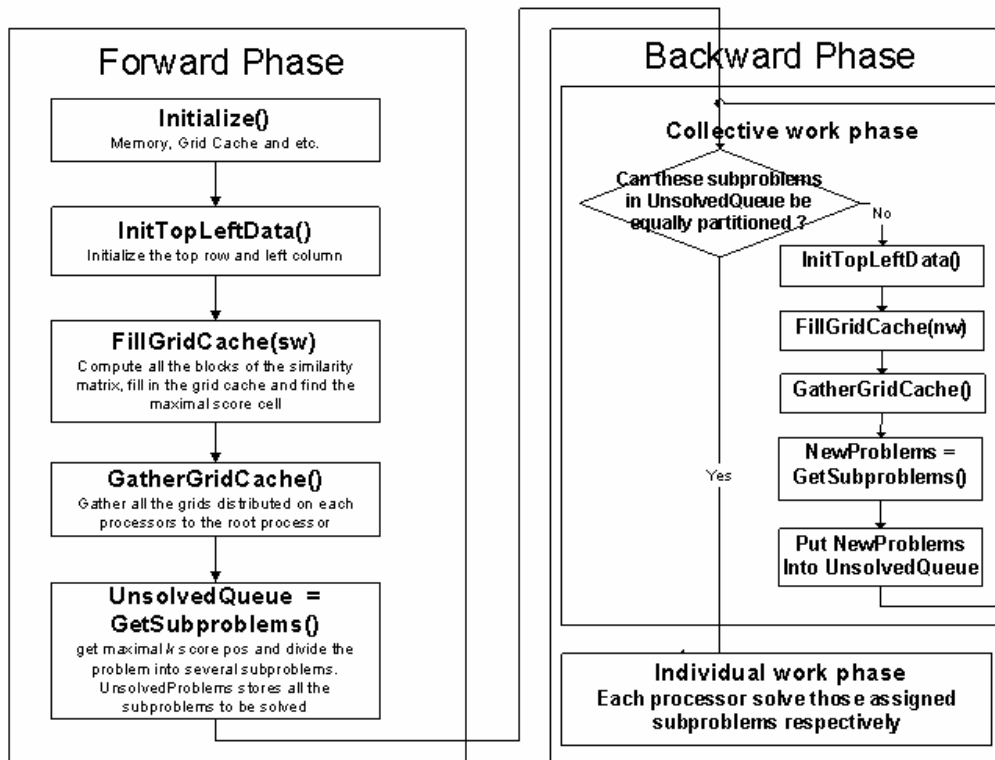


Figure 3 Flowchart of PLSA

4.1 Forward phase

In the forward phase the block performs as the elementary unit to compute the whole sequence alignment based on the full matrix filling method. The grid caches, including the position, score and local/global start point information will be recorded at the same time. Moreover, k maximal positions should be saved and updated during this period. After that, we will use this information to decompose the whole problem into a series of independent subproblems and solve them as global sequence alignment problems.

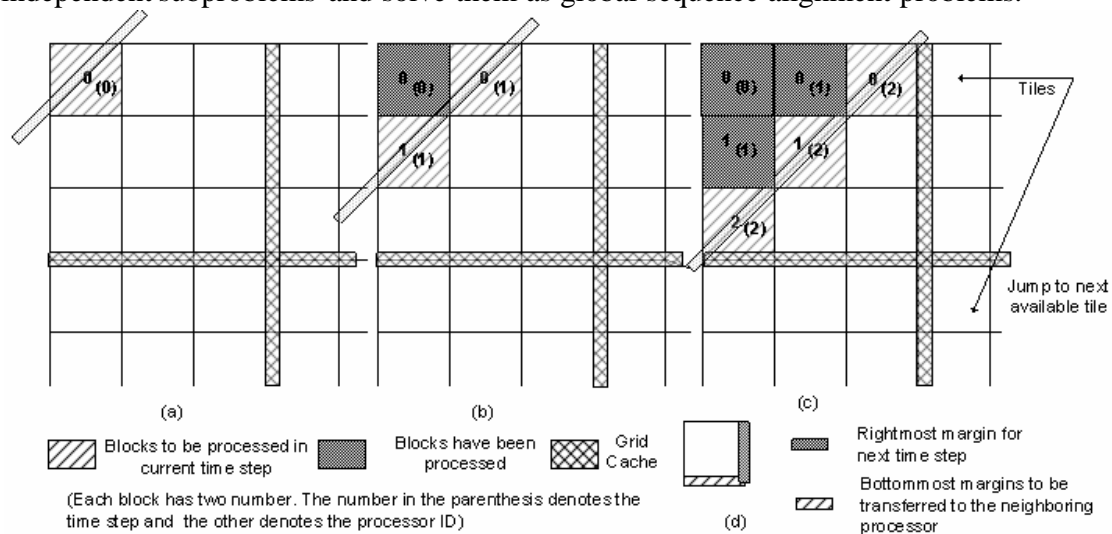


Figure 4 Wave front parallelism

The computation of each block follows the dynamic programming Eq (1) to fill the block matrix. The whole similarity matrix should firstly initialize the values of the leftmost column and topmost row by *InitTopLeftData* procedure in Fig 3. After that, the top-left block can be computed immediately. Since a block has dependencies on its adjacent left, upper-left and upper blocks according to Eq (1), it can only be processed until its adjacent related blocks being computed. Based on this dependency model, the wavefront communication pattern can be used in the parallelization of similarity matrix. The wavefront moves in anti-diagonals as depicted in Fig 4, and the shift direction is from the north-west to the south-east. Parallelization of the wavefront computation has several different ways according to the particular parallel architecture used. On fine-grained architectures such as shared memory system, the computation of each cell or a relatively smaller block within an anti-diagonal is parallelized, and this is only valid for very fast inter-processor communications since the granularity for each unit processing is extremely small. On the other hand, for distributed memory system such as a PC cluster, it is more efficient to assign a relatively larger block to each processor. In our implementation, two parameters h and w of the block size, denoting the height and width in terms of cells, can be tuned. We can simply modify these two parameters to adapt to different architectures.

In the PLSA algorithm, *FillGridCache* procedure in Fig. 4 performs block calculation, storing and updating k maximal cells and filling grid cache tasks. It is the most time-consuming module in the algorithm, taking up almost 99.5% of total execution time. In order to better exploit the data locality and minimize the communication overhead, we use tile based processing scheme in the wavefront parallelization. A tile is a bundle of blocks consisting of a complete horizontal blocks, which can be processed only on a single processor. Once a processor finishes computing a block, it will continue to process all the remaining blocks in this tile until the tile is empty. Then this processor can proceed to work on the next available tile. E.g, in Figure 4(a), since only block(0,0) has the initial values, processor 0 will work on block(0,0) at the first time step while all the other processors are idle. After that, the bottommost margin in block(0,0) is transferred to processor 1 and Processor 0 moves on to compute block(0,1) in this tile. Processor 1 uses this margin as the initial topmost margin of block(1,0) and the rightmost margin of block(0,0) are reserved as the initial leftmost margin of block(0,1). In this way, block(0,1) and block(1,0) can be computed by processor 0 and 1 simultaneously at the second time step. Similarly, P processors can process P blocks at the same time. The processing of these tiles advances on a diagonal-like front, and all the P processors can work in parallel after $(P-1)$ time steps.

Along with the block computing, grid cache should also be saved when a part of grid columns or rows is within a computing block. Since the grid cache is distributed on all the processors, we will use procedure *GatherGridCache* to collect all the distributions of the grid cache. At the same moment, the local maximal k cells in every processor are gathered and sorted to select the global maximal k score cells either. Finally, we use *GetSubProblems* procedure to find all the segmented subproblems and put them into the *UnsolvedQueue*. This subproblems queue will be processed in the backward phase.

When investigating the communication pattern between two adjacent blocks, we can observe that each block usually has two communication operations, receiving the bottommost marginal data from the upper block and sending its marginal data to its

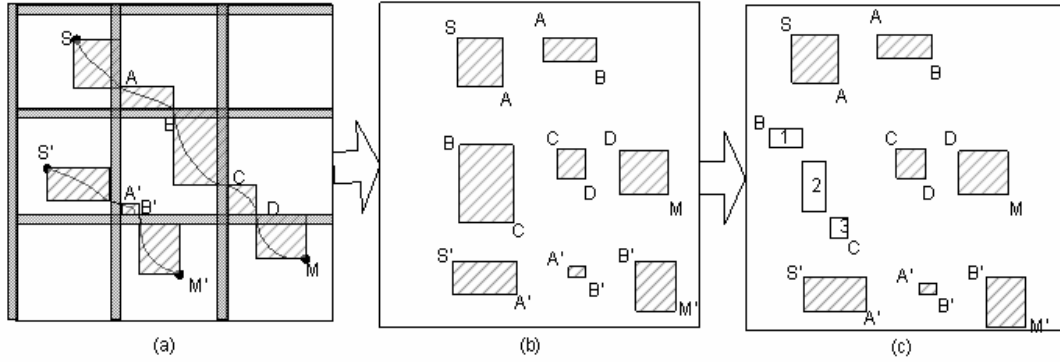
bottom block. In order to efficiently hide the communication overhead especially in a PC cluster. Non-blocking receive message passing operations [18] are used to overlap the communication overhead with computing. The whole process will work exactly like a pipeline block by block until the whole similarity matrix is filled. It greatly minimizes the communication cost and delivers better parallelization performance.

4.2 Backward phase

After the forward phase, we get a series of independent subproblems stored in the *UnsolvedQueue*. For each subproblem, we use global alignment algorithm to solve these subproblems and concatenate these alignments to the optimal path. Actually, solving the subproblem alignment can be regarded as a repeated process of forward phase. However, there are several differences between forward and backward phases as follows.

- Since the start and end point of the optimal alignment paths are unknown in the forward phase, it is necessary to use Smith-Waterman algorithm to fill the whole similarity matrix and find all the subproblems. Whereas each subproblem in the backward phase has fixed start and end point, therefore Needleman-Wunsch algorithm can be used to find global alignment of these subproblems.
- Different parallel scheme can be used in these two phases. The forward phase can only use wavefront parallelism. In the backward phase, since all the subproblems are independent with each other, we may consider more factors, such as the size of subproblems and the number of processors, or jointly combine them together to derive a better parallel scheme. Another indication is we should pay more attention to the load balance performance to efficiently use all the processors in backward period since its granularity is much finer than forward phase.
- For large scale alignment problem, in general the problem is divided into several subproblems in the forward phase. However, in the backward phase if the subproblem size is smaller than the block size, it can be directly solved by using the full matrix filling method. Otherwise, similar method in forward phase can be used to subdivide this subproblem.

Though these differences pose some challenges for parallel implementation, they provide more opportunities for us to tailor these parameters to achieve an optimal parallel scheme. The first parallelism observation is to distribute these subproblems independently over all of the processors. Each processor works on a subproblem using sequential method as section 3 described. After that, all the processors collect the sub-alignments together and concatenate them to the optimal alignment. The major drawback of this scheme is that the number of processors and subproblems may mismatch with each other and the size of subproblems varies dramatically. It will cause significant imbalance among all the processors. Another parallel indication is same as FastLSA, first it decomposes a subproblem in wavefront parallel scheme recursively until all the descendant subproblems can be solved in block size. After that, it moves to the next subproblem. Nonetheless, this scheme is too fine grained and may only fit for small scale processors. Therefore we should consider both the load balance as well as the granularity problems in backward parallel phase, and design a flexible scheme to partition these tasks equally for all the processors.



(a) Two maximal score node M and M' with their alignment paths
(b) The subproblems in the unsolved queue can not be partitioned equally on all processors
(c) The largest subproblem (determined by B,C in (b)) is chosen and decomposed to 3 descendent subproblems (1,2,3). This process proceeds until all the subproblems can be distributed to all the processors equally

Figure 5 Partition Process in collective phase

In order to better exploit load balance and problem granularity, we propose a flexible scheme to dynamically partition all the subproblems. In our implementation, the backward phase is further separated into two periods: collective solving subproblem phase and individual solving subproblems phase. During the collective solving phase, we first detect whether the unsolved subproblem queue is in “balanced state” or not. The “balanced state” means that the whole subproblems can be distributed equally to all the processors within a threshold (the default value is 20%). In another words, it indicates that the difference of the sum area of the subproblems assigned to each processor are within the threshold value. E.g., the unsolved subproblem queue consists of four subproblems with different size (100*100, 50*50, 70*70 and 110*100) to be assigned to two processors. To obtain the equivalence between these two processors, these two processors are assigned with (100*100, 70*70) and (50*50, 110*100) subproblems respectively. We can simply compute the size difference ratio on these two processors, and the value $(14900-13500)/13500 = 10.3\%$ is smaller than the default threshold. Therefore we can conclude that this unsolved subproblem queue is in “balanced state”. However, this is not always the case. When we find the unsolved subproblem queue is not in the “balanced state”, we will pop up the largest size subproblem from this queue and decompose it into several smaller descendant subproblems with wavefront parallelism. After that, these descendant subproblems are pushed back in the unsolved queue. Then a new iteration will proceed on it to detect whether it is in the “balanced state” or not. Fig. 5 demonstrates how the collective phase works. Suppose the forward phase has segmented 8 subproblems with two alignments. If we find the unsolved subproblems can not be distributed equally to all the processors, the largest block “B->C” in the scenario will be decomposed into some smaller descendant subproblems as shown in Fig 5(c). This decomposition process proceeds until all the available subproblems can be approximately equally assigned to all the processors.

After the unsolved subproblem queue is in “balanced state”, we will enter the individual solving subproblem phase. During the period, each processor is approximately assigned with equal size subproblems and solves them independently using the sequential method as illustrated in section 3. Finally, after all the subproblems have been solved, the

sub sequence alignment results should be collected and concatenated to the final optimal alignment paths.

5. Experimental results

In this section, we will summarize the experimental results of the PLSA algorithm. Firstly, we will briefly introduce the overview of the experimental results and generate several observations on the performance of the algorithm such as speedup and execution time. Secondly, a detailed analysis will be described to cover the following questions: what is the bottleneck of the implementation of this algorithm; which parameter will significantly affect the performance of the algorithm and how to tune these parameters to obtain the best performance.

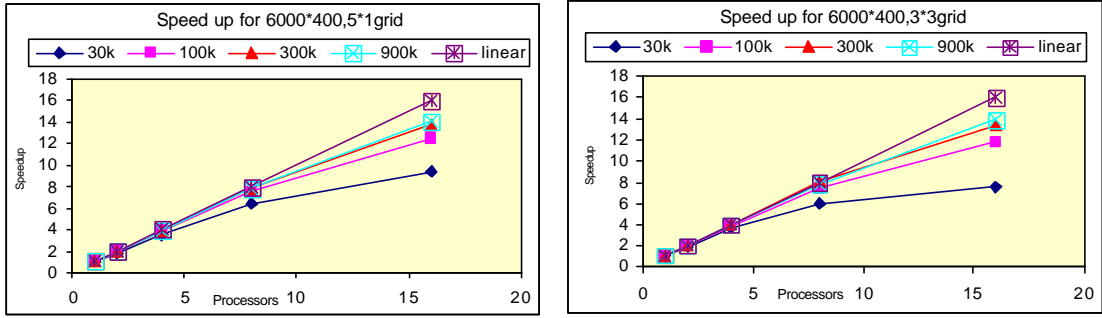
5.1 Experimental environment

The experiments of this paper are carried out on a 16-node PC cluster interconnected with a 100Mbps Ethernet switch. Each node has a 3.0GHz Intel Pentium-4 processor with 512KB second-level cache and 1GB memory. We use RedHat 9.0 Linux operation system and MPICH-1.2.5 [17] message passing library as the software environment in the experiments. All of our implementations of the PLSA algorithm are written from scratch in C++ without reference to any free smith-waterman implementations.

In order to evaluate the scalability of the algorithm, DNA sequences ranging from 30K to 900K length are chosen as test sequences for accurate scaling use. Since true sequences can seldom satisfy this specific size, we use some artificial sequences in the experiments. To make the experiments more comprehensive, we also use several real DNA sequences which are chosen from a test suite suggested by the bioinformatics group at Penn Stat University [12] covering the typical length of sequences. The longest sequence pair we used is named TCR where the human sequence is 319,030 bp long and the mouse sequence is 305,636 bp long. Fig 6 shows the speedups for this pair of sequence, and it has slightly better performance than the artificial sequences. As for other real sequences, the performances are almost the same.

5.2 General observations

The parallel characteristics of the proposed algorithm are obtained by performance analysis tools such as Intel VTune[7] and Vampir[8] as well as some theoretical analysis. There are a couple of factors affecting the program's scalability performance, such as the sequential area, communication overhead and load imbalance and so on. Fig. 6 exhibits the speedup and the execution time performance of the PLSA algorithm with 16 processors. We use 5 typical sequences in our experiments covering from 30k*30k to 900k*900k pair-wise sequences to make the experiments more comprehensive. Different parameters are adopted in our experiments, e.g., the block size is 6000*400 ($h=6000$ and $w=400$) and the grid divisions are 3*3 and 5*1 (the number of grid rows is 5 and grid columns is 1).



		Sequences Size (6000x400, 5x1 grid, 1 path)				
	Processors	30k * 30k	100k * 100k	300k * 300k	900k * 900k	TCR
Total Time (s)	1	66.9	683.1	6049.9	53523.0	6973.3
	2	36.4	345.6	3045.2	27269.1	3486.2
	4	19.0	178.1	1565.8	13897.4	1765.4
	8	10.6	90.0	765.9	6722.5	880.4
	16	7.1	54.5	435.5	3778.5	486.4
Speedup	2	1.84	1.98	1.99	1.96	2.00
	4	3.52	3.84	3.86	3.85	3.95
	8	6.31	7.59	7.90	7.96	7.92
	16	9.42	12.53	13.89	14.17	14.34

Figure 6 Execution Time and Speedup for different size sequences

For small scale sequence alignment such as 30k*30k problem, the speedup is linear for 2 and 4 processors, but starts deteriorating when 8 and more processors used. The slowdown for more processors occurs because the granularity of the work assigned to each processor decreases. E.g., the height of the block will be adjusted to $30k/8=3.65k$ in the forward phase, much smaller than the pre-assigned 6k size. And the backward phase has to decompose the subproblems into even smaller size descendant subproblems to achieve load balance for all processors. Therefore tremendous communication overhead will occur in this period.

Similar trend happens in 100k*100k sequence alignment. The speedup ascends almost linearly for up 8 processors and increases a little slow for 16 processors. The 12.5x speed up number indicates that all the processors are not fully utilized. It provides much better granularity for the paralleled tasks, but not enough to satisfy 16 processors. With the increase of the sequence alignment size, we get much better speedup curves, almost linear speedup for 300k and 900k problems, and our statistical result shows that the parallel scheme can scale well with the problem size. More specifically, 15.1x speedup can be obtained with 16 processors when outputting 3 near-optimal paths. Comparing the scalability of the forward and backward phase, we can observe that the forward phase is better than the backward phase with the increase of processors. This is not surprising since the backward phase has to decompose a number of smaller subproblems which incur tremendous synchronizations and communications overhead.

5.3 Parameter Tuning

The parameters play an important role in the application's behavior. In this section we will show how these parameters affect the performance in detail.

5.3.1 Block size

Block size (h*w)	(a) 128*32	(b) 6000*400	(c) 3000*800
Average time on 8 CPU (s)	193.3	100.0	115.0
Average time on 1 CPU (s)	706.0	743.6	743.7

Table 1 Execution time with different block size for 100k*100k alignment problem

The application behavior is closely dependent on the block size due to the impact of cache misses. If the size of block can fit into the second level cache, it will incur very few cache misses in the block computation period and get better performance. Table 1 shows that (a) is better than (b) and (c) on single processor system since the block size of (b) and (c) exceed the second level cache capability. However, when coping with 8 processors on the PC cluster, things are totally changed. This time (a) is the loser, suffers from tremendous communication penalties and runs almost double time than (b) and (c) in this scenario. Therefore we should select some larger block size to efficiently hide the commutation overhead. In additional, there are also some limitations on the block width since the communication cost go ups with the increase of the block width. If we compare the performance between (b) and (c) on 8 CPUs, we will find (b) is the winner, since the message size to be transferred in (c) is twice larger than (b). Considering both the communication cost and task granularity, generally the optimal block choice is something like a very thin rectangle with large height and small width.

5.3.2 The division of grid

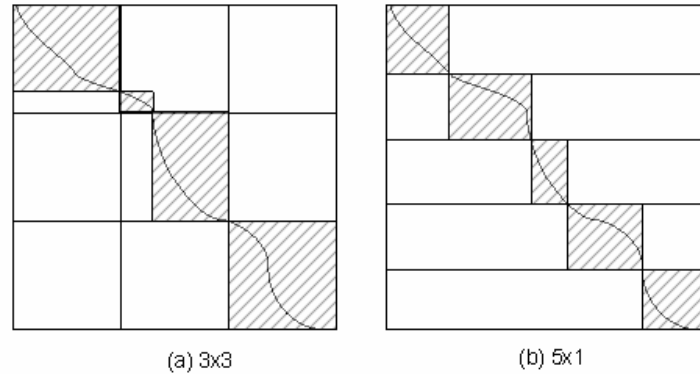


Figure 7 Grid divisions in 3*3 and 5*1 configuration

Different grid divisions may cause significant performance variations. The execution time will reduce with the increase of the number of grids. The time saving comes from the backward phase where the re-computations decrease dramatically since the total area of the decomposed subproblems becomes smaller when the number of grids increases. As an extreme case, if the number of grid caches is same as the whole problem size, PLSA becomes the equivalent to the full matrix filling algorithm without any re-computations in the backward phase. However, this is not practical due to the limitation of the memory size.

Another indication is the number of grid rows and columns may impact the whole application performance. The experiments show that in average 3*3 grid divisions is almost 15% slower than 5*1 grid divisions though both of them have the same memory requirement. Fig 7 gives an explicit explanation for this case. Since the computation of the subproblems can be regarded as the measurement of the total shaded area, the partitioned shaded area in (b) has one-fifth of the total area, whereas (a) has a nearly one-third of the total area. This indicates 5*1 will have shorter execution time than 3*3 grid

configuration. Similarly, division of 11×1 will exhibit better performance than division of 6×6 . Therefore, we can conclude that the grid divisions of $(2 \times N - 1) \times 1$ configuration is often better than $N \times N$ configuration if we have totally $2 \times N$ grid rows and columns.

5.3.3 Number of near-optimal alignments

Our proposed PLSA algorithm can output k near-optimal maximal non-intersecting alignments within one forward and backward phase. In our experiments, the speedup in k alignments ($k > 1$) is usually better than the single alignment. Since the forward execution time is relatively stable and more subproblems can be generated along with the increase of output alignments, e.g., two paths produce 8 subproblems whereas one path can only generate 5 subproblems in Fig 5(a). With more subproblems, it will be much quicker to enter the “balanced state” rather than decompose them into smaller ones recursively. Therefore we can get better parallel performance in the backward phase with more alignments. Our experiments exhibit that 15.1x speedup can be obtained with 3 paths, outperforming 14.2x speedup with only one path on 16 processors.

6. Conclusions

In this paper we propose an efficient algorithm PLSA for long sequence alignment with linear space. The basic idea is to trade space for time, where several key techniques have been put forward in our algorithm. The local start point and grid cache can divide the whole sequence alignment problem into several independent subproblems, which dramatically reduces the re-computations of traceback phase and provides more parallelism than FastLSA. In addition, global start point can benefit to find k near-optimal non-intersecting alignments. The experiments demonstrate that the parallel PLSA exhibits good scalability even on commodity PC clusters. To summarize, the combination of flexible parameter tuning, less sequential computation and much better speedup performance make the PLSA a better choice for large scale pairwise sequence alignment.

Acknowledgement

We thank Carole Dulong for reviewing this paper and providing useful feedbacks. She is our colleague from Corporate Technology Group, Intel Corporation.

References

1. Adrian Driga, Paul Lu, Jonathan Schaeffer, Duane Szafron, Kevin Charter and Ian Parsons. “FastLSA: A Fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment”. In the International Conference on Parallel Processing, 2003.
2. Aluru, S., Futamura, N., Mehrotra, K., “Biological sequence comparison using pre?x computations,” Proc. 13th IEEE International Parallel Processing Symposium, 653-659, 1999.
3. A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. Nucleic Acids Research, 27(11):2369–2376, 1999.

4. A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.
5. Chunxi Chen and Bertil Schmidt. "Computing Large-scale alignments on a Multi-luster". In the IEEE International Conference on Cluster Computing, 2003.
6. D.S., Hirschberg. A linear space algorithm for computing longest common subsequences. *Comm. ACM*, 18:341-343, 1975.
7. Intel Corp., Intel® Vtune™ Performance Analyzer, (available on-line: <http://developer.intel.com/software/products/vtune/>).
8. Intel Corp. and Pallas, <http://www.pallas.de/pages/vampir.htm>
9. Martins, W.S., del Cuvillo, J.B., Cui, W., Gao, G.R., "Whole Genome Alignment using a Multithreaded Parallel Implementation," Proceedings 13th Symposium on Computer Architecture and High Performance Computing, September 10-12, 2001.
10. Myers, E., Miller, W., "Optimal alignments in linear space," *Computer Applications in the Biosciences*, 4:11-17, 1988.
11. N. T. Perna and et al. Genome sequence of enterohaemorrhagic *Escherichia coli* O157:H7. *Nature*, 409(6819):529–533, 2001.
12. Penn State University. Bioinformatics Group. <http://bio.cse.psu.edu>, 2001.
13. Saul B. Needleman and Christian D. Wunsch. "A General Method Applicable to the Search for Similarities in the amino acid Sequence of Two Sequences". *Journal of Molecular Biology*, 48:443-453, 1970.
14. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990. (CABIOS), 4:11–17, 1988.
15. Temple F. Smith and Michael S. Waterman. "Identification of Common Molecular Subsequences". *Journal of Molecular Biology*, 147:195-197, 1981.
16. W.R.Pearson, D.J.Lipman. Improved tools for biological sequence comparison, *Proc. Natl. Acad. Sci. USA*, 85:2444-2448, 1988.
17. <http://www-unix.mcs.anl.gov/mpi/mpich/>
18. <http://www.mcs.anl.gov/mpi>