

A Million-Fold Speed Improvement in Genomic Repeats Detection

John W. Romein, Jaap Heringa, and Henri E. Bal

*Vrije Universiteit
Faculty of Sciences, Department of Computer Science
Bio-Informatics Group & Computer Systems Group
Amsterdam, The Netherlands*

{john,heringa,bal}@cs.vu.nl

Abstract

This paper presents a novel, parallel algorithm for generating *top alignments*. Top alignments are used for finding internal repeats in biological sequences like proteins and genes. Our algorithm replaces an older, sequential algorithm (Repro), which was prohibitively slow for sequence lengths higher than 2000. The new algorithm is an order of magnitude faster ($O(n^3)$ rather than $O(n^4)$).

The paper presents a three-level parallel implementation of the algorithm: using SIMD multimedia extensions found on present-day processors (a novel technique that can be used to parallelize any application that performs many *sequence alignments*), using shared-memory parallelism, and using distributed-memory parallelism. It allows processing the longest known proteins (nearly 35000 amino acids). We show exceptionally high speed improvements: between 500 and 831 on a cluster of 64 dual-processor machines, compared to the *new* sequential algorithm. Especially for long sequences, extreme speed improvements over the old algorithm are obtained.

1 Introduction

Repeats in biological sequences such as genes and proteins are the result of, and an essential mechanism for, evolution. Internal gene duplication at the DNA level allows nature to create redundant copies of genes coding for functionally important proteins. The duplication may result in enhanced expression

of the protein, or result in an improved protein, since one of the gene copies could retain the old function while the others are then free to evolve to an adapted function.

Gene duplication can take place at the level of copying complete genomes consisting of billions of nucleotides, down to only two or three nucleotides. Roughly, more than half the number of base pairs in most genomes are part of a repeat [3], which shows the importance of these copying mechanisms as a general mechanism for evolution. However, pathologically repeated fragments are also known to play a role in serious diseases like Huntington's.

Recognizing repeats in protein sequences is important, since they can reveal much information about the structure and function of a protein. Often, divergent evolution in repeats has blurred the ancestral ties, so that at first sight, the repeats hardly show any resemblance anymore. Frequently, only 10–25% of the amino acids in a repeated protein subsequence are conserved. Also, the repeats may have different lengths through insertions and deletions. Moreover, two repeats need not be consecutive (tandem), but may be interspersed by other subsequences (that can contain different repeats as well). These properties make it computationally very challenging to recognize repeats automatically.

Repro [1, 4] is an accurate method to find and delineate repeats in proteins.¹ Since it became available in 1993, it has been one of the standard and most exact methods in protein internal repeats detection [8]. However, the run-time complexity of $O(n^4)$ has thus far been prohibitive for analyzing protein

¹<http://mathbio.nimr.mrc.ac.uk/~rgeorge/repro/>

$$M_{i,j} = \max \left\{ \begin{array}{l} E_{S_{1_i}, S_{2_j}} + \max \left\{ \begin{array}{l} M_{i-1, j-1} \\ \max_{1 \leq x < i-1} (M_{i-1-x, j-1} - P_x) \\ \max_{1 \leq y < j-1} (M_{i-1, j-1-y} - P_y) \end{array} \right\} \\ 0 \end{array} \right\}$$

Equation 1: Value of an alignment matrix entry.

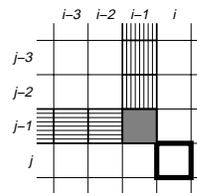


Figure 1: The boxed entry depends on the shaded entries.

sequences at the genomic scale [2, 6].² Our objective was to speed up computations “by all means”, so that even the longest known proteins (up to a length of 34350) could be processed. The Repro algorithm spends nearly all its time in finding top alignments (as discussed in Section 2.2). We reduced this time by a factor 10^6 using the following methods, which are also the new contributions described in this paper:

- We devised an $O(n^3)$ algorithm that computes exactly the same top alignments as the original algorithm, while maintaining $O(n^2)$ space complexity. The algorithm is at least 100 times faster for short sequences, and more than 1000 times faster for long sequences.
- We describe a new technique to perform parallel sequence alignment using the SIMD multimedia extensions found in present-day processors. Counterintuitively, we apply SIMD parallelism in a *coarse-grained* way, and align four (Pentium III) or eight (Pentium 4) sequences concurrently. This results in surprisingly high and superlinear speed improvements (6.9 and 9.8, respectively). The technique is applicable to *any* algorithm that performs many sequence alignments, and thus to most bio-informatics applications.
- We parallelized the algorithm even further, for shared and distributed-memory machines. Parallelizing the algorithm is challenging, since it speculatively breaks a sequential ordering in the algorithm. Using 64 dual-CPU SMPs, we obtained additional speedups between 74 and 123, compared to the SIMD version.

Biological results are outside the scope of this paper.

The paper is structured as follows. Section 2 explains sequence alignment basics and nonoverlapping top alignments; readers familiar with these subjects can skip the section. In Section 3, we explain the new sequential algorithm; Appendix A discusses

some details. Section 4 shows how the algorithm is parallelized, and Section 5 gives performance results. The last section discusses the results, concludes, and gives directions for future research.

2 Background

We will first show how a sequence pair can be aligned. Next, we will explain *nonoverlapping top alignments*.

2.1 Pairwise sequence alignments

Biological sequences (DNA, messenger-RNA, transfer-RNA, and proteins) can be compared using a technique called *aligning*. The sequences are shifted in such a way that they yield an optimal score when superposed (written out on top of each other), given some score metric that values the similarity of two individual sequence elements (the *exchange matrix*). The exchange matrix contains high scores for two identical or similar sequence elements, and low or negative scores for unrelated ones. *Gaps* may be introduced to introduce better alignments; however, for each gap, a (gapsize-dependent) penalty is subtracted from the total score. *Global alignment* compares entire sequences; *local alignment* concentrates on the subsequences that match well. In this paper, we only align locally.

Consider two example sequences: **CTTACAGA** and **ATTGCGA**. For this example, we use a simplistic exchange matrix that awards two points for matching elements and subtracts one point for different elements. We also subtract two points for each new gap (*gap opening*), and one point times the length of the gap (*gap extension*). The optimal local alignment for this sequence pair is:

```

T T A C A G A
| | | | |
T T G C - G A

```

²Reference [4] mistakenly reports an $O(n^3)$ time complexity.

and has a score of 6: 5×2 (matches) $- 1 \times 1$ (mismatch) $- (2 + 1 \times 1)$ (gap). Since we perform a *local* alignment, the initial mismatching prefixes **C** and **A** are omitted; their presence would lower the total score.

Sequences can be (globally) aligned using a dynamic programming algorithm introduced by Needleman and Wunsch [7]. Smith and Waterman slightly changed the algorithm to perform local alignments [13]. For sequences S_1 and S_2 , the local alignment algorithm first computes an alignment matrix with scores that are computed according to Equation 1. Each matrix entry $M_{i,j}$ is computed as follows (see also the boxed entry in Figure 1). First, it is decided whether or not to create a gap in one of the sequences. This is achieved by taking the maximum of $M_{i-1,j-1}$ (the gray-shaded entry, meaning no gap), the row left of it (the horizontally-shaded entries, introducing a gap of length x in the horizontal sequence) while subtracting the proper gap penalty P_x , and the column above it (the vertically-shaded entries, introducing a gap of length y in the vertical sequence) while subtracting the gap penalty P_y . To this maximum, we add the (match or mismatch) value from the exchange matrix E . In *local* alignments, negative values are disallowed, thus we maximize the value with 0, yielding the new matrix value.

	C	T	T	A	C	A	G	A
A	0	0	0	2	0	2	0	2
T	0	2	2	0	1	0	1	0
T	0	2	4	1	0	0	0	0
G	0	0	1	3	0	0	2	0
C	2	0	0	0	5	0	0	1
G	0	1	0	0	0	4	4	0
A	0	0	0	2	0	4	3	6

Figure 2: Local alignment matrix for **CTTACAGA** and **ATTGCCGA**.

Figure 2 shows the alignment matrix of our example sequence pair, where the highest matrix entry has a score of 6. From this entry, we perform a *traceback* to construct the alignment, in *reverse* order. The traceback searches the matrix how the highest score was established in the direction of the upper left-hand-side corner. The circled entries in the figure show the traceback for the example alignment.

Figure 3 shows pseudo code for the computation

of the alignment matrix. If only the alignment score is desired and no traceback is performed, there is no need to store the entire matrix; the last computed row is sufficient, significantly reducing the memory requirements. Several memory-efficient algorithms exist that do perform a traceback using only a linear amount of memory (at the expense of extra computations), but these are not covered here.

2.2 Nonoverlapping top alignments

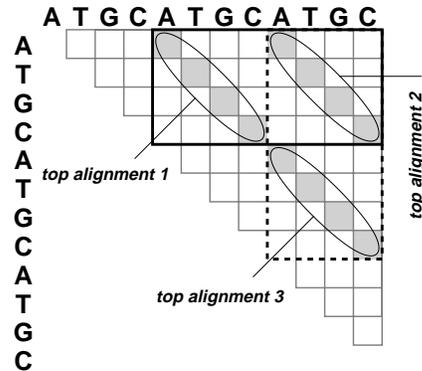


Figure 4: Three nonoverlapping top alignments.

Top alignments are used to delineate internal repeats. We explain the concept of a *nonoverlapping top alignment*, using the example of Figure 4, which shows three top alignments for the sequence **ATGCATGCATGC**. Basically, we split the input string into two pieces in each possible way, and match the first part with the second part, looking for the best matches. The figure shows 11 partially overlapping rectangles (alignment matrices). All rectangles touch the upper right-hand-side corner, but have unique lower left-hand-side corners along the main diagonal. To easily locate them, two rectangles are shown with bold, black lines (one solid and one dashed); the others are gray. Each rectangle represents a split input string with the prefix depicted vertically and the suffix depicted horizontally. Matching the prefix **ATGC** (indicated by the top four rows) with suffix **ATGCATGC** (the rightmost eight columns, which together form the black, solid rectangle) yields the first two, equivalent top alignments, namely **ATGC**·**ATGCATGC** and **ATGC**·**ATGCATGC** (circled in the figure). The third (also equivalent) top alignment is found when matching prefix **ATGCATGC** with suffix **ATGC**, yielding top alignment **ATGCATGC**·**ATGC** (note that this rectangle can also produce the second alignment). Top alignments 1 and 3 together form

```

PROCEDURE ComputeMatrix(Seq1, Seq2, ExchMat, GapOpenPenalty, GapExtPenalty) IS
  FOR y IN 1 .. Seq1.Length() DO
    FOR x IN 1 .. Seq2.Length() DO
      M[y][x] := MAX(0, ExchMat[Seq1[y]][Seq2[x]] + MAX(MaxX, MaxY[x], M[y-1][x-1]));
      MaxX := MAX(M[y-1][x-1] - GapOpenPenalty, MaxX) - GapExtPenalty;
      MaxY[x] := MAX(M[y-1][x-1] - GapOpenPenalty, MaxY[x]) - GapExtPenalty;
    END;
  END;
END;

```

Figure 3: Pseudo code for the function ComputeMatrix().

the strongest “signal” but are separate top alignments since their concatenation is not nonoverlapping. This can be seen in the figure: there is no surrounding rectangle that encloses the concatenation of 1 and 3. More details on top alignments are given in Section 3. The Repro method uses the top alignments to delineate the repeats. Some tens of top alignments are required; more top alignments increase Repro’s sensitivity.

3 The new sequential algorithm

The sequential algorithm computes a user-defined number of top alignments, typically 10–30, some more for large sequences. The first top alignment is computed as follows. We define $S_{a:b}$ as the substring of S that starts at position a and ends in position b . There are $m - 1$ ways to split sequence S of length m into two subsequences $S_{1:r}$ of length r and $S_{r+1:m}$ of length $m - r$. Each subsequence $S_{1:r}$ is aligned locally with $S_{r+1:m}$. Using the example of Figure 4, we would first align **A** with **TGCATGCATGC**, then **AT** with **GCATGCATGC**, and so on. Splitting S into disjoint parts guarantees that aligned fragments do not overlap.

We thus align $m - 1$ pairs. The alignment that has the highest score constitutes the first top alignment. We only compute the score of each alignment, but do not yet perform a traceback to construct the alignment (since no traceback is performed, it is not necessary to store the entire alignment matrix; the previously computed row is sufficient). We store the score of each alignment for later use, so that we know which alignment yielded the highest score after we aligned all subsequence pairs. As explained in Appendix A, we store the bottom row for later use as well.

After the first top alignment is found, we continue searching subsequent alignments. New top alignments are not allowed to overlap with alignments already found. We therefore realign in such a way that we prohibit alignments over the matrix entries

that correspond to matched amino acid pairs that are already part of an existing top alignment. This can be achieved efficiently by overriding matrix entries when computing new alignment matrices: if a matrix entry is already contained in a top alignment, its value is set to zero, otherwise it is set to the value that Smith-Waterman would have normally produced. In Figure 4, for example, this would mean that once a field is shaded gray, the corresponding matrix entries in all matrices (rectangles) that contain the field are overridden with zero in subsequent realignments. Note that other entries may change as well: overriding a matrix entry often causes a cascade of entries towards the right and the bottom to be lowered, since these entries frequently depend (indirectly) on the just overridden entry. A triangular matrix with boolean values, which we call the *override triangle*, is used to keep track of the entries that are contained in a top alignment. Each time a top alignment is found, the alignment is traced back, and the entries in the override triangle that correspond to the reconstructed path are set to “true”. The override triangle contains $m \times (m - 1) / 2$ entries. Since the triangle is sparse, it can be compressed if memory usage is an issue. The appendix explains how we reject false shadow alignments that are rerouted artificially around an already-existing alignment.

Except for the first top alignment, it is not necessary to realign *all* subsequences to find the next nonoverlapping top alignment. We order the realignments in such a way, that we realign the most promising ones first, as explained below. For each subsequence pair (rectangle), we maintain the best score that was found after its most recent alignment, as well as the number of top alignments already found at that time. The latter signifies with which override triangle the most recent alignment was performed. The score that resulted from a previous alignment with an outdated override triangle is an upper bound for the score that will be obtained when the pair is realigned with the current override triangle (since the new override triangle overrides

```

1 PROCEDURE FindTopAlignments(NrTopAlignmentsNeeded)
2   FOR  $i$  IN 1 .. Sequence.Length() - 1 DO
3     Task. $r$  :=  $i$ ;
4     Task.Score := INFINITY;
5     Task.AlignedWithTopNum := -1;
6     InsertTask(Queue, Task);
7   END;
8
9   WHILE NrTopAlignmentsFound < NrTopAlignmentsNeeded DO
10    Task := GetTaskWithHighestScore(Queue);
11
12    IF Task.AlignedWithTopNum == NrTopAlignmentsFound THEN
13      INC(NrTopAlignmentsFound);
14      TracebackAndUpdateOverrideTriangle(Task);
15    ELSE
16      Task.Score := AlignWithoutTraceback(Task);
17      Task.AlignedWithTopNum := NrTopAlignmentsFound;
18    END;
19
20    InsertTask(Queue, Task);
21  END;
22 END;

```

Figure 5: Pseudo code for the function FindTopAlignments().

more entries, the new score will generally be lower, and never be higher). The scores of the previous alignments are used as ordering heuristic. We repeatedly select the subsequence pair with the highest score from its most recent alignment. If its most recent alignment was with an outdated override triangle, it is realigned with the current override triangle. Otherwise, there are no subsequence pairs that could yield a better score, thus we accept the pair as new top alignment, and update the override triangle. This way, we prevent many realignments that provably cannot yield a satisfactory score; it typically reduces the number of realignments by 90–97%.

The way we order realignments is illustrated in Figure 5. We maintain a task queue where the tasks are ordered with respect to their scores. Each task has a value r that identifies the subsequence pair (line 3), a *Score* that either represents an upper bound on the (re)alignment score before the (re)alignment is done, or the real score after the (re)alignment is done (line 4), and a value *AlignedWithTopNum* that indicates when (and thus with which override triangle) the most recent (re)alignment was done (line 5). Initially, each score is set to infinity and last alignment number to -1, to reflect the fact that the corresponding subsequence pair has never been aligned. All tasks are entered into the queue (line 6). Since all scores are infinity, the order is unimportant; each task is aligned once anyway before the first top alignment is accepted. After initialization, the most promising pair

is repeatedly selected (line 10). It is accepted as top alignment (lines 13–14) if it has already been aligned with the current override triangle. Otherwise, it is (re)aligned (line 16), and the fact that it is now aligned with the current override triangle is registered (line 17). In all cases, the task is requeued (line 20), at a position that depends on its score. The entire process is repeated until all required top alignments are found.

The new algorithm is quite different from the old one. The main difference is the introduction of overriding zeros, enabling a run-time complexity improvement from $O(n^4)$ to $O(n^3)$.

4 Parallelism

We deploy parallelism at three different levels: using SIMD multimedia extensions, at shared-memory level, and at distributed memory level.

4.1 SSE/SSE2 parallelism

At the lowest level, we use the multimedia extensions³ found on present-day Intel and Athlon processors to obtain four or eight-fold parallelism. These SIMD-style instruction set extensions were intended to vectorize simple loops, i.e., to allow

³This term is widely used but in fact a misnomer; it refers to a particular application domain, while its application area is much broader.

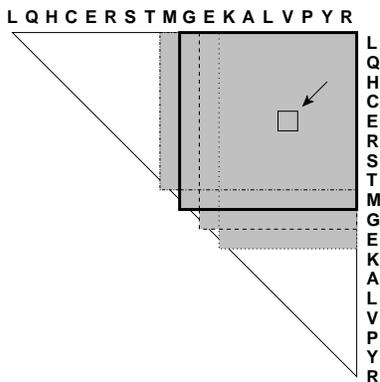


Figure 6: A matrix with three neighbors.

concurrent computations on independent array elements. Alignment scores are typically integer values that fit in short (2 byte) integers. The Pentium III and Athlon processors, with their SSE instructions, can perform a single operation on four shorts simultaneously, while the Pentium 4, with its SSE2 instructions can perform an operation on eight shorts at the same time. To some extent, a vectorizing compiler (such as Intel’s) can automatically generate parallel code from a sequential program, provided that it can assure that the array computations in a loop are data independent. Additionally, *compiler intrinsics* allow the programmer to explicitly use the multimedia extensions without writing assembly. One can declare 8 or 16-byte variables of multimedia data types just like a variable of any other type, and perform operations on them by “function calls” that are recognized and treated specially by the compiler. Although the programmer more or less specifies which multimedia instructions to use, the compiler relieves the programmer from the burden of register allocation and instruction scheduling.

It is difficult to parallelize Needleman-Wunsch-style matrix computations. Normally, the matrix is computed row (or column) wise, but a loop-carried dependency (MaxX in Figure 3) disallows using such fine-grained parallelism. It is possible to compute the entries diagonally, from the left or lower border to the right or upper border, such that all entries in a diagonal can be computed independently, but the administrative overhead is large. Rognes and Seeberg describe an SSE implementation that obtains a six-fold speedup by speculatively breaking the aforementioned data dependency [9, 10]. However, they operate on 8 bytes rather than 4 short integers, increasing the amount of parallelism, but lim-

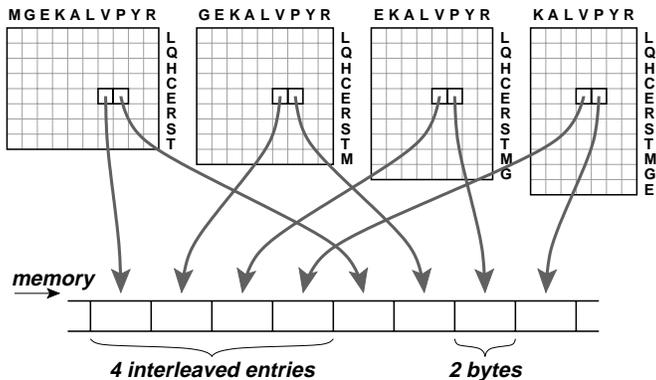


Figure 7: Interleaving matrix entries in memory.

iting the maximum score to 255. For medium and large-sized sequences, this limit is too restrictive.

We apply SIMD-style parallelism at a coarser grain size, and compute four independent alignment matrices at the same time (or eight using SSE2; in the text below we assume SSE with four-fold parallelism). Each time a matrix is to be computed (since it appears at the head of the task queue), we compute three “neighboring” matrices as well (see Figure 6). The matrices are quite similar to the one that needs to be computed: they only have a few columns less or more at the left-hand side and a few rows less or more at the bottom side. Due to their similarity, the scores in the matrices and the final results are also similar. Moreover, if a matrix is scheduled for computation (because it has the highest current score), it is likely that the neighboring matrices will be scheduled for computation shortly thereafter, since their scores are probably nearly as high. Thus although we speculate on the necessity to compute the neighboring matrices, the odds are that they have to be computed anyway.

The four matrices are computed concurrently, hardly changing the sequential algorithm to align one pair of sequences. The order in which the entries are computed is the same as in the sequential algorithm, but now we apply SIMD operations to all corresponding entries in each matrix, e.g., the scores for the entries in the figure that align **V** and **E** (indicated by the arrow in Figure 6) are computed at the same time. Since multimedia instructions operate on consecutive memory entries, we *interleave* the matrix entries in memory, as illustrated by Figure 7. For our application, the corresponding entries align the same amino acids (or nucleotides), and thus use the same exchange value from the exchange matrix. If

we were to align four unrelated pairs of sequences, we had to look up the distance between each pair of amino acids separately, therefore we align neighboring matrices, so that the amino acids are the same. Another advantage of using similarly-sized matrices is that corrections for the left and bottom borders are easily made.

Each matrix has a fixed set of neighbors. The matrices are grouped in subsequent groups of four, i.e., group 1 contains matrices 1–4, group 2 contains matrices 5–8, and so on. We schedule groups of matrices in the task queue, rather than individual matrices. The matrix with the highest score in the group determines the score of the entire group, and the task queue is ordered with respect to the groups' scores.

When programming with the multimedia extensions, memory bandwidth easily becomes an issue. Since the processor performs more operations per cycle than with conventional programming, more data flows through the memory bus. Our alignment routine is *cache-aware*, and runs mostly in first-level cache. Remember that only the previously computed row is stored, and that the entries of the four simultaneously computed matrices are interleaved over the memory. We achieve cache-awareness by computing the matrix in vertical stripes: instead of computing one row after another, we compute a section of the row that fits in a third of the first-level cache size, after which we compute the section of the row below it. Another third of the first-level cache is used for storing the corresponding section of the MaxY array (see Figure 3), leaving the last third for miscellaneous data like the exchange matrix. The increase in speed easily compensates the administrative overhead incurred at the stripes' boundaries (see Section 5.1).

The compiler intrinsics allowed a simple implementation without writing assembly. We implemented the parallel alignment routine for both SSE and SSE2-enabled processors.

4.2 Shared-memory parallelism

The static parallelization scheme we use for SIMD-style parallelism (computing a fixed set of neighbor matrices) does not scale to hundreds of matrices. When group members become more distant, the matrices become more dissimilar and their scores will differ increasingly. If the scores of some group members are much lower than that of the highest-scoring group member, the speculative parallelism becomes less efficient; the chance increases that the best-scoring member of a suboptimal-scoring group is better than the worst-scoring members of

the best-scoring group. As a consequence, inferior matrices are computed (concurrently with the most-promising matrix) while other high-scoring matrices are waiting in the task queue. Another disadvantage of very large fixed groups is that more corrections have to be made to the left and bottom borders.

For MIMD-style parallelism, we therefore use a dynamic task-scheduling system, that schedules multiple jobs from the task queue ahead. When a thread is idle, the parallel scheduler selects the task with the highest score from the task queue that has not already been assigned to another thread. After the task has been completed, the scheduler reenters the task (with its new score) in the task queue. Like in the sequential algorithm, a new top alignment is found if the scheduler finds a task at the head of the queue that has already been aligned with the current parameters. The parallelism is speculative (albeit more efficient than the static scheme described in Section 4.1): if one task results in a new top alignment, the other running tasks are not of interest anymore. Nevertheless, the work for the superfluous tasks is not wasted. Since their associated scores are usually lowered, they are reentered further back in the queue, and often they are not considered anymore for quite a long time.

An advantage of using multiple threads is that the large, main data structures are shared in memory. The parallelism is coarse grained, and the amount of time spent in critical sections is negligible. Our implementation uses the Posix threads library.

4.3 Distributed-memory parallelism

The third level of parallelism is applied at the distributed-memory level. We deploy the same dynamic scheduling scheme as the shared-memory scheduler. For portability, we use MPI to distribute work. However, MPI is designed for statically-partitioned data parallelism, and is less suitable for applications that do not know when they can expect an incoming message, such as Repro.

To fit the application in the MPI programming paradigm, our implementation sacrifices one processor (the master) that manages the task queue and provides all other processors (the slaves) with work. The master hands out a task to an idle slave, which, after the work on the task has finished, sends back the results to the master, that reenters the task into the queue. The master waits for incoming messages if it has nothing else to do. Sacrificing one processor assures that work requests and replies from slaves are handled instantaneously; although MPI supports asynchronous communication, it does not provide a

mechanism for interrupting a thread upon message receipt, so that a thread can suspend its normal computations to handle an incoming message immediately.

The use of MPI also impacts data-partitioning decisions. Obviously, we replicate the override triangle, since all processors frequently read the data and seldomly update it (only when a new top alignment is found). Less obvious is the partitioning of the last-row data that is computed when a matrix is scheduled for the first time (with an empty override triangle; see Appendix A for details). Our implementation stores all last rows on the master processor; other processors that need the data can ask the master processor for a replica (once computed, the last row data never changes), and cache it. If the master does not have enough main memory to store all last rows (this becomes an issue if sequences are longer than 40000, requiring 1.5 GB memory), such a scheme is not possible. In this case, an exclusive partitioning scheme is needed where each processor stores part of the last rows. A processor that needs the last row would then either need to ask the owner processor to send it, or should migrate the remaining work to the processor that owns the last row (the benefits of asynchronous work migration are described in [11, 12]). However, the owner processor does not know when it can expect such communication, and is required to continuously poll the network to process incoming messages instantaneously, which would significantly increase the communication overhead.

Our implementation supports a combination of shared and distributed-memory machines, i.e., a cluster of SMPs. Although it is possible to start multiple independent processes on a single shared-memory multi-processor that communicate through MPI, this wastes much memory, since each SMP keeps multiple copies of all shared data structures (one for each process on the SMP). Therefore, we run multiple threads on each SMP that share these data structures. A small complication is that thread support is not integrated with our MPI implementation, therefore we protect all MPI calls with a mutex. If the master processor resides on a SMP, the other processors are regular slaves.

5 Performance

In this section, we show performance results for our algorithm. The test set includes the largest known protein, human titin, with a sequence length of 34350 amino acids. *The final version of this pa-*

per will provide numbers for more sequences.

The measurements are performed on DAS-2, a computer cluster with 72 nodes, where each node is equipped with 2 Pentium III processors running at 1.0 GHz, and at least 1.0 GB of main memory. The system is connected through Myrinet, a 2 Gb/s bidirectional, switched network. For the SSE2 tests, we used a Pentium 4 at 2.53 GHz. We used the Intel 7.0 C compiler, optimizing for the respective targets.

length	old (s)	new (s)	speedup
1000	1121	10.6	106
1200	2460	17.6	140
1400	5251	28.4	185
1600	8347	42.3	197
1800	14672	57.4	256
⋮	⋮	⋮	⋮
34350	?	343000	?

Table 1: Run times on a Pentium III for the old and new sequential algorithm.

Table 1 compares run times for the old and the new sequential algorithm on a Pentium III. These are the times needed to compute 50 top alignments for the first n amino acids in titin. The new algorithm is more than a factor 100 faster than the old one, and the speedup increases quickly with the sequence length, due to the order of magnitude difference in run-time complexity. The estimated runtime of the old algorithm for the *entire* sequence clearly exceeds any reasonable time limit. Extrapolation of the column with speedups indicates that the new algorithm is thousands of times faster.

For the parallel performance, we show speedups with respect to our sequential implementation of our *new* algorithm. Test input that is sufficiently large to be interesting for parallel processing using our new algorithm does not finish in a reasonable time using the (sequential) old algorithm; test input that is sufficiently short for the old algorithm runs too fast using the new sequential algorithm to justify parallel processing.

A single alignment computation is coarse grained; depending on the size of the matrix, the sequential implementation needs up to 5.2 seconds for the largest matrices (17175×17175) on the Pentium III, and 2.7 seconds on the Pentium 4.

5.1 SSE/SSE2 parallel performance

Table 2 shows (maximum) alignment times. The column “conventional” gives times for alignments

	conventional	SSE	SSE2
Pentium III	5.2 / 1	3.0 / 4	—
Pentium 4	2.7 / 1	1.8 / 4	2.2 / 8

Table 2: Maximum alignment times in seconds. “3.0 / 4” should be read as “three seconds to align four sequence pairs.

using the conventional instruction set. SSE gives times to compute 4 matrices, and SSE2 to compute 8.

Using the multimedia extensions, we obtain excellent speedups: $6.9 (= \frac{5.2}{3.0/4})$ for a Pentium III using SSE. The speedups for a Pentium 4 are 6.0 when using SSE and 9.8 when using SSE2. At this speed, more than a billion matrix entries per second are computed.

The speedups are much higher than may be expected (even superlinear), since SSE performs 4 and SSE2 performs 8 simultaneous operations. We mention a few causes for the exceptionally high speedups. First, the SSE and SSE2 extensions contain a (parallel) MAX operator,⁴ which is not available in the conventional instruction set. For each matrix entry, five MAX operations are required; the conventional instruction set requires several instructions to perform a single MAX operation. Second, SSE and SSE2 provide new sets of 8 (mmx/xmm) registers, decreasing the pressure on the small conventional register file and reducing the need to spill registers to memory. Third, the compiler and processor schedule instructions in such a way that the conventional instructions (e.g., used for controlling loops) and the SSE/SSE2 instructions (used for computing matrix entries) are executed concurrently in the execution engines, maximizing the use of computational resources.

When using SSE, the cache-awareness of the alignment routine significantly increases the alignment speed; depending on the dimensions of the matrix, cache-aware alignment is up to 6.5 and on average about 4 times as fast as alignment without striping. For alignments using the conventional instruction set, cache-aware alignment is also faster, but by a marginal 16%.

We also observe that the speculation overhead for our application is small: the SSE version hardly computes more alignments than the sequential version (less than 0.70%). The total runtime of the SSE version is 6.8 times as low as the sequential version.

⁴Unfortunately, it is only available for signed shorts and unsigned bytes, not for unsigned shorts or any other integer type.

5.2 Multi-processor performance

Using the second CPU in a dual-processor machine yields a 100% performance increase; since the algorithm is coarse grained and runs nearly entirely in the first-level caches, the processors run nearly independently (the latter is not true for the noncache-aware algorithm: contention on the memory bus limits the speed increase to merely 25% when the second CPU is used).

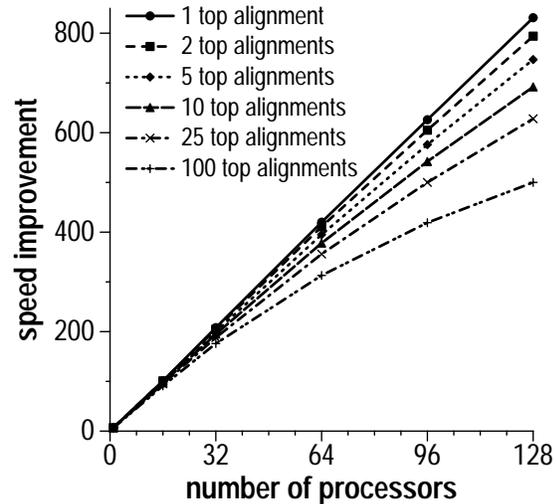


Figure 8: Speed improvements for computing up to 100 top alignments for titin.

Figure 8 shows speed improvements for titin using up to 128 processors. The improvements for finding the first top alignment are nearly perfect. For 128 processors, we measured an improvement of 831. With respect to the SSE version (on a single CPU), we obtain a speedup of 123. The efficiency thus is $123/128 = 96.1\%$. The 3.9% performance loss is caused by sacrificing one master processor and by a small load imbalance at the end of the iteration, since the traceback of the top alignment is done sequentially and takes a relatively long time. Communication overhead is small; each slave processor sends up to 64 KB/s, and neither the master processor nor the Myrinet network forms a bottleneck.

After having found the first top alignment, the speedups decrease. This is because there is not enough parallelism to keep all processors busy, not even speculatively. Usually, only 3–10% of the matrices need realignment with a new override triangle before the next top alignment is found. The decreased speedups are partially caused by the speculative scheduler, that started work that turned out to

be irrelevant (up to 8.4% more alignments were performed than by the sequential algorithm), and partially caused by idle slave processors. On 128 processors, we still obtain a 500-fold speed improvement with respect to the sequential version, and a 74-fold speedup compared to the SSE version.

6 Discussion, conclusions, and future work

This paper illustrates the extraordinary achievements that can be obtained by multi-disciplinary research. We presented a new, parallel algorithm that computes top alignments for the well-known 10-year-old *Repro* method, that accurately delineates repeats in biological sequences like proteins and genes. Using 128 processors, the new algorithm is a million times faster than the old sequential algorithm, and enables processing of much longer sequences than formerly, including the longest known protein. Most of the performance gain was obtained by reducing the (sequential) run-time complexity of the old algorithm from $O(n^4)$ to $O(n^3)$; for short sequences, a performance improvement of more than a factor of 100 was obtained, which rapidly increases for longer sequences. Extrapolating the run times yields improvements of a factor of thousands for long sequences.

We parallelized the new algorithm at three levels: using SIMD multimedia extensions, shared-memory parallelism and distributed-memory parallelism. Especially the use of multimedia extensions resulted in surprisingly good and superlinear speed improvements (6.9 on a Pentium III, 9.8 on a Pentium 4). Combined, the three levels of parallelism resulted in speed improvements between 500–831 on a cluster of 64 dual-CPU Pentium IIIs. In total, the new parallel algorithm processes long sequences easily a million times faster than the old sequential one.

We claim that the way we perform parallel alignment using multimedia extensions is also applicable to other application areas that require many alignments, and thus to many bio-informatics applications. A prerequisite is that the matrices have more or less the same dimensions. In practice, alignments are usually performed on similarly-sized sequences. In contrast to our application, the general case requires looking up exchange values sequentially, slightly decreasing the parallel performance. However, the use of the instruction set extension will still give good speedups — without using any extra hardware.

Future research includes work on the second part of the *Repro* method, that delineates repeats using the top alignments that have been computed. Although that part works very well for small sequences, it needs some changes to increase the sensitivity for long sequences, such as extra filtering to select the “best” repeat (in a sequence **AACAACAACAAC**, is it better to delineate two occurrences of **AACAAC**, four occurrences of **AAC**, or eight occurrences of **A** ?), and more tuning to find the “right” starting positions of tandem repeats (since repeats are usually quite different from each other, the boundaries are often vague).

We are confident that with the achieved speed increase and the preserved sensitivity of the old *Repro* algorithm, many thus far undiscovered genomic repeats will be recognized, aiding the understanding of genome evolution and repeat-associated diseases.

Appendix

A Algorithm details of the sequential algorithm

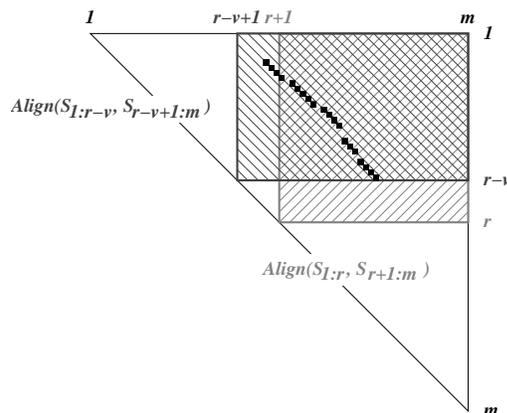


Figure 9: Top alignment ends in row $r - v$.

This appendix explains some details of the sequential algorithm described in Section 3. One of those details is the following observation. Since we perform *local* alignments, alignments do not necessarily end in the bottom (last) row or rightmost column. Yet, we only have to check the scores in the *bottom row* of each alignment matrix to find the top alignment, saving many tests to find the maximum score and, as we will see later in this appendix, much memory. This fact is not intuitive and is explained now. Suppose that a particular local alignment, e.g.

A_r of length r and B_r of length $m - r$, does not end in the bottom row, but v rows above the bottom row (see Figure 9, where the chain of black pixels represents a top alignment). Since we compute the alignments of all subsequences of S , we will also align A_{r-v} and B_{r-v} . The latter alignment is rather similar to A_r and B_r , and therefore yields a similar alignment matrix, albeit with v fewer rows at the bottom and v more columns prefixed on the left. It is highly likely that the matrix contains the same alignment, that now *does* end in the last row. If it does not yield the same alignment, its score must be even better, since it has a few columns prefixed that might even improve the alignment. Anyway, the top alignment will end in one of the matrices' bottom rows, therefore we only have to check the bottom rows of all matrices to find the top alignment.

The new algorithm overrides matrix entries with zeros. Waterman and Eggert [14] also published an algorithm that overrides matrix entries with zeros; Huang et al. [5] followed their approach with an algorithm that reduced the memory requirements from $O(n^2)$ to $O(n)$. However, our algorithm rejects shadow alignments, as explained below. When overriding matrix entries with a zero, we create alignments that avoid the overridden entries. Sometimes, these alignments are suboptimal: their scores would have been better if they were allowed to use the overridden entries (this happens when the tail of the alignment is in the cascade of matrix entries that were lowered by overriding some matrix entries with zero; see Section 3). Even though such an alignment is suboptimal, its score can be so high that it would be accepted as a new top alignment. Yet we would not like to accept it as a top alignment, because they are artificially rerouted along already found alignments. To avoid accepting such shadow alignments, we only accept an alignment if it has the same score as it would have had when computed without the overridden entries. There are several ways to check this. It is possible to align the subsequences with and without an override triangle, and use the alignment that yields the best, *equal* score in both cases (unequal scores signify shadow alignments). This is computationally expensive, since it requires aligning each pair twice, except before having found the first top alignment, since the override triangle is empty then. Another way is to actually use the latter fact. After a matrix is aligned for the first time (thus with an empty override triangle), we store the bottom row. We only have to store the bottom row, since we only check bottom rows for ending positions of top alignments. In subsequent realignments, we compare the bottom row of the re-

alignment with the original (stored) one: unequal values signify shadow realignments and are invalid. The score of the best realignment is the maximum of the valid entries in the bottom row. Storing all bottom rows requires $m \times (m - 1)/2$ memory (also a triangle), and is the largest data structure that we use. For practical problems, it will typically fit in main memory, but if it does not, caching or swapping is feasible. Again, on-demand recomputation of the last row is also possible at the expense of extra work; this would allow an implementation that requires only a linear amount of memory, provided that the override triangle is stored sparsely. We have, however, not found the need to implement this.

References

- [1] R.A. George and J. Heringa. The REPRO Server: Finding Protein Internal Sequence Repeats Through the Web. *Trends in Biochemical Sciences*, 25(10):515–517, October 2000.
- [2] A. Heger and L. Holm. Rapid Automatic Detection and Alignment of Repeats in Protein Sequences. *Proteins: Structure, Function, and Genetics*, 41:224–237, 2000.
- [3] J. Heringa. Detection of Internal Repeats: How Common are They? *Current Opinion in Structural Biology*, 8:338–345, 1998.
- [4] J. Heringa and P. Argos. A Method to Recognize Distant Repeats in Protein Sequences. *Proteins: Structure, Function, and Genetics*, 17:391–411, 1993.
- [5] X. Huang, R.C. Hardison, and W. Miller. A Space-Efficient Algorithm for Local Similarities. *Computer Applications in the Biosciences*, 6:373–381, 1990.
- [6] E.M. Marcotte, M. Pellegrini, T.O. Yeates, and D. Eisenberg. A Census of Protein Repeats. *Journal of Molecular Biology*, 293:151–160, 1998.
- [7] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48:443–453, 1970.
- [8] M. Pellegrini, E.M. Marcotte, and T.O. Yeates. A Fast Algorithm for Genome-Wide Analysis of Proteins with Repeated Sequences. *Journal of Molecular Biology*, 35:440–446, 1999.

- [9] T. Rognes. ParAlign: a Parallel Sequence Alignment Algorithm for Rapid and Sensitive Database Searches. *Nucleic Acids Research*, 29(7):1647–1652, 2001.
- [10] T. Rognes and E. Seeberg. Six-Fold Speedup of Smith–Waterman Sequence Database Searches Using Parallel Processing on Common Microprocessors. *Bioinformatics*, 16(8):699–706, 2000.
- [11] J.W. Romein and H.E. Bal. Solving the Game of Awari using Parallel Retrograde Analysis. *IEEE Computer*, 2003. *Accepted for publication*.
- [12] J.W. Romein, H.E. Bal, J. Schaeffer, and A. Plaat. A Performance Analysis of Transposition-Table-Driven Scheduling in Distributed Search. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):447–459, May 2002.
- [13] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
- [14] M.S. Waterman and M. Eggert. A New Algorithm for Best Subsequence Alignments with Application to tRNA–rRNA Comparisons. *Journal of Molecular Biology*, 197:723–725, 1987.