ASM

DNA Assembly Application

Jacek Błażewicz et al.

RA – 001/ 2004

Authors:

Prof. Dr. Habil. Jacek Błażewicz

Ph.D. Marta Kasprzak

B.Sc. Przemysław Jackowiak

B.Sc. Dariusz Janny

B.Sc. Dariusz Jarczyński

B.Sc. Maciej Nalewaj

B.Sc. Bartosz Nowierski

B.Sc. Rafał Styszyński

B.Sc. Łukasz Szajkowski

B.Sc. Paweł Widera

CONTENTS

1. Introduction

1.1. Motivation

The molecular biology is a science domain having an outstanding practical meaning and the funds invested in it bring considerable profits. Nowadays, the research in a field of genetics is evolving very intensively. In 1995 a whole genome of a popular bacterium Escherichia Coli was read [1]. The next year's achievement was reading of the yeasts genome [2]. The triumphal march of the genomics continues while more and more organisms' genomes become known. Not surprisingly, the scientists very soon desired to reach the Holy Grail – the human genome. The parts of it were already known, especially the genes responsible for diseases. However, an exact recognition of the human genome was a distant goal. In spite of that, the genome research engaged thousands of scientist inspired by the future profits: a better understanding of the inheritance process, an early disease discovery, more efficient vaccinations and therapies. The vision also tempted the pharmaceutical industry and governments – the main sponsors of the research. The race for the supremacy in the area of genome reading has started.

The first success in this domain was announced by the Celera Genomics Company on 26th June 2000. Celera claimed it had read the whole human genome [3]. However, it was proven to be very inaccurate and so the race continues. The main racer is Human Genome Organization (HUGO) held by about 1200 researches from 40 countries [4] [5].

To achieve the goal, an interdisciplinary cooperation between biologists and computer scientists is necessary, as the computer scientists play a significant role in modern molecular biology. The mix of these two scientific areas is called computational biology. One of the problems considered by the computational biology is DNA assembly. It deals with sequences of millions of base pairs (bp) long, or even longer (like the human genome, which is believed to have about 3 billion bp). The traditional computing methods are not sufficient to analyze the data of such a size. To solve the problem there is a need of fast computers and smart heuristic algorithms. That is why one of the first researches over DNA assembly was undertaken in Los Alamos (New Mexico) [6] on its powerful supercomputers. The discovery of the precise human genome is still a challenge amongst computational problems.

The subject of analysis in this report is the DNA sequence assembly. An assembly is a process of composing the long DNA fragments from many short ones through the

analysis of their common (overlapping) parts. It makes, obviously, no difference whether to assembly the DNA of a human, a horse, a mouse or an Escherichia Coli bacterium. Every time it is the same complex computational problem. It arises, because available biological methods are primitive and allow to read the genetic material only in short fragments (sequences 100 – 1,000 base pairs long). This process is known as sequencing [7] [8] [9] [10]. Because the short pieces of DNA are not satisfactory material for biologists, the assembly is indispensable. It constructs long fragments from the cuttings being a result of sequencing. Obtained material is further processed to find the positions of fragments in a genome. This process is called mapping and is based on restriction maps, where examined fragments of sequences are placed according to positions of restriction points (markers). However, it is said that a well made assembly can eliminate the mapping process. This is the postulate forced by former president of Celera Genomics – Craig Venter [11]. Besides, the assembly is more universal than mapping as here is no need for a map (so any, even an absolutely new sequence, could be assembled). All this proves that the DNA assembly is one of the most important problems in the modern computational biology.

Nowadays, there are several solutions of the assembly problem available. While browsing WWW sites a few computer applications trying to solve it were found. In the European Bioinformatics Institute [12], being a part of the European Molecular Biology Laboratory (it associates many scientific laboratories from 15 European countries), a JESAM package was found. It can work in a distributed environment. It is available with a source code. The Genome Center from the University of Washington [13] offers the Phrap application [14], which is one of the most widely used assembly software. Although it is free for academic use, it is pretty expensive for commercial users (a $10,000 fee must be paid for the package of Phred, Phrap and Consed applications). TIGR (The Institute for Genomic Research from Rockville) [15] presents the open source TIGR Assembler application on its website. The University of Arizona introduced a free FAKtory package [16], which contains FAKII (Fragment Assembly Kernel). Another assembly application is CAP3 [17], offered by Michigan Technological University. The application is completely free and easy to use, but very slow. Unfortunately, CAP4 [18] (the successors of CAP3, which is most likely much faster) is not publicly available. The last application is Gap4, which is a part of another open source package – Staden [19]. It offers a graphical user interface for several platforms. Apart from its own algorithm, it also allows for using external assembly applications (CAP3, FAKII or Phrap). For the

sake of comparison with the application presented in this report only Phrap and CAP3 were useful. We were unable to separate the assembly core from other considered applications, thus it was impossible to make automated tests.

1.2. Details of the assembly problem

The input data for the DNA assembly problem is a set of short sequences of up to about 1,000 bp obtained as a result of the sequencing stage (as mentioned in the section 1.1). The problem of DNA assembly is solvable only when all regions of the original sequence are covered by a sufficient number of input sequences (coverage of 6-10 times is recommended) and their positions in the original sequence are mostly different (i.e. the sequences are shifted with regard to each other). Thus, input sequences overlap with each other – this information is crucial for a process of reconstruction of the original sequence. Note that the input sequences do not have to cover the original sequence uniformly.

1.2.1. Ideal instance

An ideal instance of the assembly problem is such that every piece of the expected output sequence (original sequence) is covered by several input sequences. Moreover, it is assumed that input sequences do not contain errors and that the original sequence does not contain long repeated regions. The example of an ideal instance and the solution for it is illustrated in Figure 1.

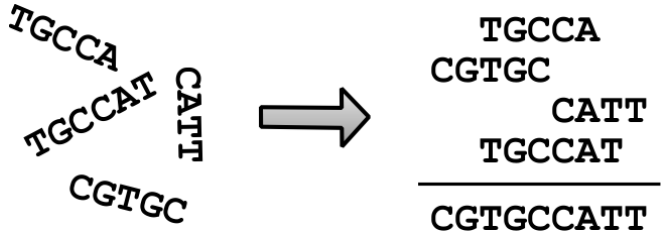

Figure 1. DNA assembly problem – ideal instance.

1.2.2. Real-life obstacles

Real-life instances are far from being ideal. Usually they are erroneous or introduce problems of another type. Such an instance may have the following properties:

- **lack of coverage** – among the input sequences there are no sequences covering some regions of the original sequence (see Figure 2);

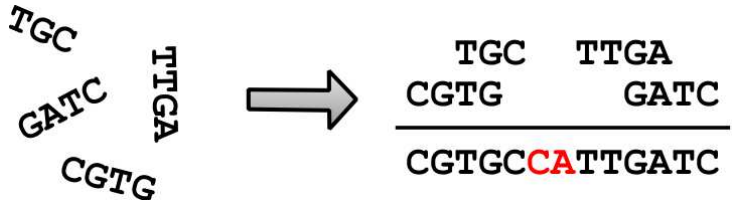

Figure 2. DNA assembly problem – lack of coverage.

- **errors:**
  - inaccuracy of sequencing process
    insertion, deletion and substitution of nucleotides (see Figure 3);

Figure 3. DNA assembly problem – inaccuracy of sequencing.

  - chimeras
    erroneous fragments, which arise when two fragments from distinct parts of the original sequence join end-to-end;
  - contamination
    fragments of DNA from another organism, which falsely got among input sequences during a process of replication;

- **unknown orientation** – it is not known from which of two DNA strands the sequence comes from – a sequence and its reverse complement become equivalent (see Figure 4);

Figure 4. DNA assembly problem – unknown orientation.

- **repeated regions** – long fragment of a sequence, which are identical or very similar.

1.3. Project objectives

The aim of this project is to make a DNA assembly service available to users through the Internet at so called *bioportal* (in a sense of a portal concerning biology), developed as a part of the PROGRESS Project [20]. The computations are performed on fast multiprocessor machines sponsored by the SUN company.

The main goal of the first part of the project was to design and implement efficient, sequential (i.e. single-process) algorithms solving the DNA assembly problem. The algorithms were developed basing on the ones described in Dr. Marta Kasprzak's report [21]. The second part of the project was the implementation of a distributed version of the algorithms, taking the advantage of the available multiprocessor machines.

The scope of the project was to implement two variants of the assembly algorithm, further referred to as *the ideal case* and *the real case*. The ideal case assumes that there are no errors in the input instances, while the real case takes into account the inaccuracy errors (see the section 1.2.2). In both cases, the algorithms should be aware that a lack of coverage may occur and that sequences may come from both DNA strands. Obviously, allowing the errors causes the problem to be more difficult. Therefore, the real case algorithms are more complex and more computationally expensive.

As a result of the project, a distributed assembly application was created. Further in this report, it will be referenced to as ASM.

1.4. Cooperation

When creating the ASM application, our center has cooperated with universities and organizations all over the world. The contact with scientist from areas of biology, chemistry as well as computer science was essential to create a good, usable assembly algorithm. Furthermore, without them the DNA assembly application would have no practical justification, as they are the ones who will use it. This was the reason for starting cooperation with many scientists all over the world. Below is the list of people who are potentially interested in using our application or were helpful in the process of its creation:

- Prof. Dr. Habil. Wojciech T. Markiewicz, Insitute of Bioorganic Chemistry, Poznań;
- Prof. Dr. Habil. Marek Figlerowicz, Insitute of Bioorganic Chemistry, Poznań;
- Prof. Dr. Habil. Jerzy Tiurym, Warsaw Univeristy, Warsaw;
- Dr. Pablo Moscato, The University of Newcastle, Australia;
- Prof. Dr. Martin Vingron, Max Planck Institute for Molecular Genetics, Berlin;
- Prof. Dr. Rudolf Amann, Max Planck Institute for Marine Microbiology, Bremen;
- Dr. Agnes Szczepek, Max Planck Institute for Infection Biology, Berlin;
- Todd Taylor, RIKEN Genomic Sciences Center, Yokohama;
- Marie-France Sagot, INRIA Rhône-Alpes.

When cooperating with Dr. Pablo Moscato, we were asked to confirm the correctness of the SARS-CoV genome (the deadly coronavirus causing Severe Acute Respiratory Syndrome), assembled by Canada's Michael Smith Genome Sciences Centre [22] [23]. The shotgun data for the experiment were also available at their website. ASM was executed on these data and a few good quality sequences were obtained, covering 99.8% of the genome. After manual finishing, basing on the expert knowledge of Prof. Marek Figlerowicz, the genome was constructed. It almost perfectly matches the genome assembled by Canada's Michael Smith Genome Sciences Centre, thus its quality was confirmed.

2. Algorithms

This section describes roughly the algorithms which were used in the presented application. The section 3 shows the way they were implemented for a distributed grid environment.

The ASM application consists of two methods. Both are heuristic. The one described in the subsection 2.1 deals with the ideal case – it is extremely fast, but its usage is limited, because it is not realistic. The second method described in the subsection 2.2 deals with the real case.

2.1. Algorithm for the ideal case

In the ideal case, it is assumed that input sequences are errorless, so sequences which are next to each other in the original sequence must overlap perfectly (suffix of one must match exactly the prefix of another one). However, not all overlaps are taken into account. Algorithm takes on input a parameter *mo* (***M****inimum* ***O****verlap*). It denotes minimal number of nucleotides on which two sequences must overlap so that this overlap could be regarded significant (values less than 10 are not recommended; 15-30 should be enough in most cases). In this way, accidental overlapping of input sequences, which do not come from the same region of the original sequence, is limited.

The algorithm is composed of four stages. The first stage identifies overlaps between sequences and creates an overlap graph, being a directed multigraph. This is the most complicated and time-consuming stage, thus a good design and an efficient implementation required a lot of work to have been done. The second stage reduces the size of the constructed graph by deleting arcs, which are unnecessary and could disturb further computations. It also calculates a reliability score for remaining arcs, based on their relation with the deleted ones. The third stage is a heuristic trying to produce a directed Hamiltonian path in the graph, the most reliable as possible, which represents the solution. The last stage is the construction of the consensus sequence from the path found in the previous stage, which is trivial in the ideal case.

2.1.1. Overlap graph construction

During this stage a hash function is used [24]. The input of a hash function is a character string of a fixed length (called here: window). The output of the function is a non-

negative, integer number (called: characteristic number). Obviously, the same strings have the same characteristic numbers. Unfortunately, many different strings also can have the same characteristic number. It does not matter what kind of hash function is used – any good hash function will do.

The first and very important thing in this stage is determining of window size used by the hash function. This size is dependent on *mo* parameter. Mostly, the window size equals *mo*. When *mo* is too big, then the window size is set to smaller, more optimal value. If *mo* is very small, then using window size of that length would result in lots of hash conflicts, which would drastically slow down the computations. In such a case, a technique called double hashing is used (details are described later in this section).

The next step is calculation of characteristic numbers for the first windows of every input sequence. In order to be able to search quickly for characteristic numbers, they are memorized in a hash table (together with the number of a sequence they were obtained from).

After these preliminary steps, an overlap graph construction may begin. In such a graph each input sequence is represented by a vertex. There is an arc from vertex *v* to vertex *u* when sequences corresponding to these vertices overlap (precisely, suffix of *v* matches prefix of *u*), with regard to *mo* parameter. In order to find overlaps, a characteristic number is calculated for each window of every sequence. Suppose that a characteristic number of a window of the sequence *v* appears in the hash table then and this number belongs to the sequence *u* (i.e. it is characteristic number of the first window of *u*). Then (and only then) it is possible, that *v* overlaps *u*. Only for such pairs of sequences it is thoroughly checked (character by character) whether they overlap. When they do, an arc from *v* to *u* is added into the graph and is assigned a number denoting a shift between sequences (e.g. `ACGGACT` and `GACTCATT` overlap with shift 3). This information is important at later stages.

Additionally, when checking if two sequences overlap, we may find out that in fact one sequence is a subsequence of another one. This information is memorized in an array of subsequences and the subsequence is not taken into account when creating the overlap graph, i.e. there is no vertex in the graph corresponding to it (such vertices could distract the stages of arc reduction and finding path). Obviously, there is no more need to perform calculations sequences when they turn out to be subsequences, which can speed up the algorithm. On the other hand, if some computations have been already done, and some

arcs incident to such vertices have been created, they will be removed at the end of this stage.

As mentioned earlier, when *mo* parameter is too small, the double hashing technique is used. An additional, auxiliary hash table is created. In the main hash table there are memorized characteristic numbers calculated for the first windows of some bigger size, for which the algorithm acts the most efficiently (let it be called *ows* – ***O**ptimal **W**indows **S**ize*). In the auxiliary hash table there are memorized characteristic numbers of the first windows of size equal to *mo*. When finding overlaps, for each sequence, all possible characteristic numbers are calculated with the use of the window of *ows* size. Then, characteristic numbers for windows size equal to *mo* are calculated, but only for the final area of a sequence (i.e. last *ows*-1 characters). For example: for a sequence `ACCTGTTA` with the $ows = 5$ and $mo = 3$, first characteristic numbers for `ACCTG`, `CCTGT`, `CTGTT` and `TGTTA` are calculated and then for `GTT` and `TTA`. Of course, when using characteristic numbers for *ows* windows size we must refer to the main hash table, but when using *mo* windows size – to the auxiliary hash table.

2.1.2. Arc reduction

The arc reduction stage removes the arcs, which duplicate some information. An arc between vertices (sequences) v and w with shift s (denoted as $v \rightarrow_s w$) is deleted if there exist two other arcs $v \rightarrow_p u$ and $u \rightarrow_q w$ such that $s=p+q$ (note, that this is so called transitive arc). It would mean that information about sequences v and w overlapping with shift s, can be reconstructed from the information carried by the smaller arcs, thus, it is redundant.

Additionally, it is easy to notice, that arc $v \rightarrow_s w$ confirms that arcs $v \rightarrow_p u$ and $u \rightarrow_q w$ are not accidental, thus, making it more reliable. Therefore, we assign a score to each arc, which is a measure of its reliability. Initially, each arc has score equal to 1. When an arc $v \rightarrow_s w$ is removed, its score is added to scores of $v \rightarrow_p u$ and $u \rightarrow_q w$.

Arcs for deletion are considered in a decreasing order of shifts to avoid disposal of arcs, which are necessary to delete some other arcs (with greater shifts).

2.1.3. Finding path

Now, the algorithm searches for the best possible path through previously constructed and reduced graph. In the best case it is a Hamiltonian path and thus all input sequences are included in the output, assembled sequence. In fact, an attempt is made to construct a

Hamiltonian path with the maximum overall score (in case there are many paths of maximum score, the one producing the shortest sequence is preferred). The problem is widely known as Traveling Salesman Problem [25]. It is NP-hard, therefore a good heuristic is necessary here.

If the construction of one connected path is not possible, the output of this stage is a set of disjoint paths, hence the output of the entire algorithm is a set of sequences (so called: contig), instead of one.

2.1.3.1. Finding first element

As the first element of the constructed path, it is desired to choose such a vertex which has unattractive incoming arcs and thus it is not likely to be in the middle of a good path. Mathematically speaking, for each vertex *v* we find an incoming arc *a(v)*, which has the greatest score or the greatest overlap (i.e. the smallest shift value) in case of a tie. Then, from among all the vertices, the one with the smallest score of *a(v)* (or the smallest overlap value in case of a tie) is selected to be the first in the constructed path. Once such a vertex is selected, it is not to be used again.

2.1.3.2. Finding next elements

Having a part of the path determined, new vertices are appended one by one. In each step the potentially best successor for the last vertex of the path is chosen in the following way. Suppose that the last vertex of the so far constructed path is *v*. For each potential successor *u* a function *f(v,u)* (defined below) is evaluated and a vertex with the highest value is chosen.

$$f(v,u) = \frac{w(v,u)}{\lim 1(v)} + \frac{w(v,u)}{\lim 2(u)}$$

w(v,u) is the maximum reliability score among arcs from *v* to *u*. Value of *lim1(v)* is the maximum score of an arc beginning in *v* and ending in any other vertex. Value of *lim2(u)* is the maximum score of an arc ending in *u* and beginning in any other vertex. So, a specific compromise between the best arc outgoing from the last node in the constructed path and the best arc incoming to a potential successor is reached. In case of equal values of function *f*, a vertex which has the best arc incoming from *v* with the greatest overlap is chosen (in order to make the output sequence shorter).

It is possible that the last vertex of the so far constructed path has no successor, despite the path is not yet finished (i.e. it does not have all the vertices). This may happen

if there is no coverage or a low one at some region of the original sequence, or when the algorithm gets distracted by poor-quality data. In such a case, a construction of the path is left apart. A new, disjoint path is started and then constructed in the same way.

2.1.3.3. Reordering

The situation described at the end of the previous subsection could happen as well due to imperfect selection of the first vertex (which in fact should be somewhere in the middle of a correct path). Therefore, after creating all the disjoint paths we try to reorder them and check whether or not they fit with each other. If there is an arc from the last vertex of a path constructed later to the first vertex of a path constructed earlier, then the paths are reordered and merged into one.

2.1.4. Output sequence construction

As already mentioned, having a path (or paths) creating the output sequence (or sequences) is trivial. Each path represents exactly one contig. It can be directly reconstructed from sequences represented by consecutive vertices of the path, shifted according to shift values associated with arcs of the path.

2.2. Algorithm for the real case

In this case, the algorithm is a modification of the heuristic described in the section 2.1. This time, however, it admits errors in the input sequences in the form of insertions, deletions and substitutions.

Apart from ***M****inimum* ***O****verlap* parameter, already known from the ideal case, the algorithm takes another one – *eb* (***E****rror* ***B****ound*). This is a maximum acceptable error (ranging from 0 to 1) occurring between overlapping fragments of sequences. Error is a function of number of matching and mismatching nucleotides in overlapping fragments, as well as the length of an overlap – it will be described in details in the subsection 2.2.1. However, due to big time-optimizations, these parameters are considered only in a weak sense. To be more precise, overlaps shorter than *mo* and with error greater than *eb* are not accepted, however, not all overlaps which satisfy these limits take part in the solution construction. This simplification barely affects quality of results, but it allows to seriously decrease the time of computations.

2.2.1. Alignment

Before the algorithm would be described thoroughly, a few words about the alignment problem should be said [8] [26] [27]. The alignment is a means of comparison of sequences. Informally speaking, two sequences (or their parts) are placed next to each other and "stuffed" with spaces so that they look as similar as possible. Such an arrangement is evaluated by rewarding matching nucleotides as well as penalizing mismatching ones and spaces. In the presented algorithm, the following values are used: +1 for a match, –1 for a mismatch and –2 for a space. The *global alignment* is considered in the context of entire sequence comparison. The *semi-global alignment* is a modification of the *global alignment* where spaces at the end of the first sequences and at the beginning of the second sequence are not penalized. Thus, suffixes of the first sequence are compared with prefixes of the second sequence – this is the way the overlaps between sequences are identified.

Having the alignment and its evaluation (denoted as *score*), we calculate the *error* (in order to compare it to *eb*) by normalizing the *score*, using the formula:

$$error = (1 - \frac{score + length}{2 * length}),$$

where *length* denotes a number of overlapping nucleotides. The best possible alignment score is equal to *length* (it happens when the overlap is perfect) and the worst to *–length* (it happens if no match is found; note that in such case it is better to substitute nucleotides and there is no need to insert spaces).

2.2.2. Overlap graph construction

During a construction of the overlap graph, an alignment algorithm being a modification of the Smith-Waterman alignment algorithm [28] with addition of appropriate pruning (further in this paper it is referred to as *diagonal-bounded Smith-Waterman alignment algorithm*) and Lipman-Wilbur fast alignment algorithm [29] are used. The idea of the proposed alignment algorithm benefits mainly from two observations:

- *Observation 1:* if two sequences overlap with low error, then the path representing the best alignment in the Smith-Waterman matrix goes mainly through some diagonal (or diagonals) and its neighborhood.

- *Observation 2:* overlaps not shorter than $\frac{1}{eb}$ must have a perfectly matching region of length at least $\frac{1}{eb}-1$ (they are usually much bigger or there are many of them if an overlap is long enough).

Let a region of a sequence of some fixed size be called a window and let *ws* (**W**indow **S**ize) be its size. The value of *ws* should be more or less equal to $\frac{1}{eb}-1$. (We cannot always stick strictly to this formula, as too big values could cause bad results and too little values could cause algorithm to drastically slow down.) The idea is to compare all the windows of all the sequences of size *ws* with each other. If *ws* is large enough this can be done very efficiently using a hashing technique, in a similar way to the one described in the section 2.1.1. This time, however, all the windows (not only the first) are memorized in the hash table and no double hashing technique is used (only *ws* is used).

Knowing which windows are equal (i.e. match exactly), the step of finding overlaps can be drastically reduced. There is no need to align every pair of sequences (which is very time-costly), only pairs of sequences with a significant number of equal windows need to be aligned, because only these pairs have a chance to overlap. *Observation 2* says, that by doing so overlaps which satisfy *eb* are not missed (unless they are too short, but skipping short overlaps turned out not to be a big problem in practice). Additionally, information about equal windows could be used to prune computations of Smith-Waterman alignment algorithm. When windows of two compared sequences match, this match is marked in the appropriate place in the Smith-Waterman matrix. If there are many common windows for the two sequences and the sequences overlap well, then by *Observation 1*, many of such marks are along some specific diagonal (or diagonals). Now, it is enough to fill the matrix only in a fixed-width neighborhood of the diagonal(s), when running Smith-Waterman algorithm. This is what we called diagonal-bounded Smith-Waterman algorithm. It computes the alignment in time proportional to the length of longer of the aligned sequences. It is much faster than original Smith-Waterman algorithm, which computes the alignment in time proportional to the product of lengths of the aligned sequences.

Now, the procedure of creation of an overlap graph may be finally explained (it is somewhat similar to the one described in the section 2.1.1, with regard to the new way of finding overlaps). As in the ideal case, in such a graph each input sequence is represented

by a vertex. There is an arc from vertex *v* to vertex *u* when sequences corresponding to these vertices overlap. Obviously, not every overlap is taken into account, but only the feasible ones, i.e. of length not less than *mo* and error value not greater than *eb* (as mentioned earlier, this method is a heuristic, so some overlaps satisfying these constraints might not be taken into account). Both, the overlap length and the error, can be obtained after the alignment is done from the Smith-Waterman matrix. Additionally, the error value and a shift between overlapping sequences (which can be calculated from lengths of sequences and the overlap) are associated with every arc. Consider an example with 4 input sequences: `(1) ACTTAGTC`, `(2) AGTCCATG`, `(3) TTGTCCA` and `(4) CCAAGACT`. To see all the feasible overlaps and some infeasible overlaps refer to Figure 5 (penalized positions are marked with small letters). Figure 6 presents the graph obtained from these overlaps.

FEASIBLE OVERLAPS	
`(1) ACTTAGTC----` `(2) ----AGTCCATG`	*overlap* = 4 (*shift* = 4) *error* = 0
`(1) ACTTaGTC--` `(3) --TT-GTCCA`	*overlap* = 6 (*shift* = 2) *error* = 0.25
`(1) ACTTaGTC--` `(3) ---TtGTCCA`	*overlap* = 5 (*shift* = 3) *error* = 0.2
`(2) AGTCCAtG---` `(4) ---CCAaGACT`	*overlap* = 5 (*shift* = 3) *error* = 0.2
`(3) TtGTCCA--` `(2) -aGTCCATG`	*overlap* = 6 (*shift* = 1) *error* ≈ 0.17
`(3) TTGTCCA-----` `(4) ----CCAAGACT`	*overlap* = 3 (*shift* = 4) *error* = 0
`(4) CCAAGACT-----` `(1) -----ACTTAGTC`	*overlap* = 3 (*shift* = 5) *error* = 0
INFEASIBLE OVERLAPS	
`(3) TTGTCCA-------` `(1) ------ACTTAGTC`	*overlap* = 1 < *mo* *error* = 0
`(4) CCAGAcT-----` `(2) ----AgTCCATG`	*overlap* = 3 *error* ≈ 0.33 > *eb*

Figure 5. Example: feasible and infeasible overlaps.

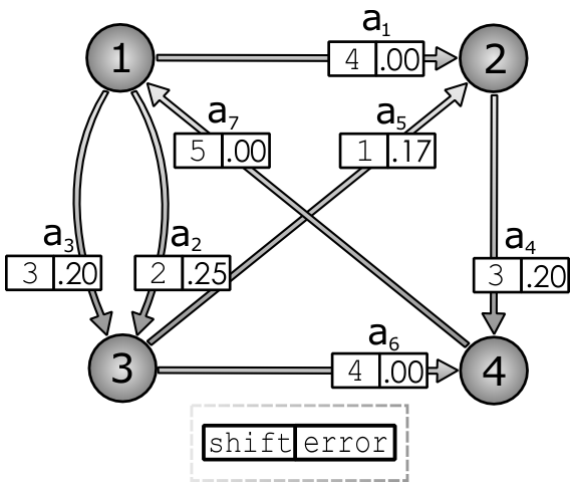

Figure 6. Example: overlap graph.

Furthermore, as a byproduct of the alignment, we obtain information about inclusion of sequences in other sequences (of course, with regard to *eb*). Exactly as it is described in 2.1.1, subsequences are memorized in a subsequence array and not taken into account when creating a graph (or removed if they are already there, along with incident arcs).

2.2.3. Arc reduction

The arc reduction stage in the real case is exactly the same as in the ideal case (described in the section 2.1.2).

2.2.4. Finding path

The stage of finding path in the real case is very similar to the one described in the section 2.1.3. However, in this case, error value must be taken into account. It is used as the first tie-breaker when reliability score (in case of the first element) or function *f* (in case of next elements) does not point a winner. Of course, error should be maximized for arcs in the procedure of determining the first element and minimized in the procedure of determining next elements. In case the error value also does not give a unique winner, the second tie-breaker is the criterion used for the ideal case, i.e. the shift value.

2.2.5. Output sequence construction

The path(s) found in the overlap graph is an input to this stage, which is not so trivial this time. Again, each path represents exactly one contig. A conting is constructed as a consensus sequence that contains all the sequences (vertices) from the corresponding path. To build the consensus sequence, input sequences are iteratively merged into one big metasequence. Contrary to alignments done in the first stage of the algorithm, this

method requires only one alignment computation per sequence. However, since overlaps between sequences are not perfect, it requires the metasequence to keep some extra data associated with each position; it is the information on how many symbols confirm each position, where a symbol is either a letter representing a nucleotide or a space. The consensus sequence is then established according to the majority rule – a nucleotide which appears most frequently on a given position is selected to the consensus, or the position is skipped in case the space is the most frequent symbol.

Merging a sequence with the so far merged metasequence requires a calculation of their alignment to determine the best merge layout. Diagonal-bounded Smith-Waterman alignment algorithm is used again. This time, however, to select the best diagonal, the algorithm benefits from the knowledge of shift between two consecutive sequences on the path.

2.3. Two-strand version

Both the heuristics, described in the sections 2.1 and 2.2, do not handle the problem of unknown orientations of input sequences. They treat sequences as if they were from one DNA strand, while they may come from two. This would results, at best, in solving problem of finding two sequences (reverse complementary to each other), each having twice less coverage than one could expect. To avoid this problem both the heuristics where extended to handle two-stranded data. This can be enabled by additional parameter – *ts* (***T****wo* ***S****trands*). This section describes small modifications introduced to both the algorithms (in both cases the changes are the same).

At the very beginning, before the first stage of the algorithm even starts, for every input sequence its reverse complement is created and added to the set of input sequences. It cannot be determined, which of the two orientations is the correct one, so the new sequences are treated equally as their originals. They are also treated independently in the overlap graph construction, arc reduction and output sequence construction stages. Only in the finding path stage, they are coupled with the originals they were obtained from.

During creation of a path, when a vertex (a sequence), either an original or a new one, is selected to be in the constructed path, its reverse complementary counterpart cannot be used later. Thus, in fact we do not look for a Hamiltonian path now. The goal is to find a path, which at best includes half of the nodes and includes only one vertex from each pair of coupled vertices (of course, we still want to maximize the score of such a path). In fact,

by selecting half of the nodes all the input sequences (either in their straight or reverse complementary form) are included in the consensus sequence.

The two-strand version affects also the reordering procedure. When a path constructed later is tried to be matched to the beginning of a path constructed earlier, both of its orientations must be checked.

The overlap graph construction, arc reduction and output sequence construction stages do not need to be modified. They completely ignore the fact, that reverse complements have been added.

3. Distributed computations

Although the algorithms presented in the previous section are quite efficient, they can be performed even more efficiently by parallelizing the computations. Therefore, a distributed version of the described algorithms was created. The models of distribution of the algorithms are thoroughly discussed in the subsections 3.2 and 3.3.

3.1. System architecture

3.1.1. System physical layout

The whole distributed system, ASM was developed on, consists of several parts. The central part is formed by three SUN Fire 6800 servers equipped with fast Sparc CPUs. Two of them are located in Poznań Supercomputing and Networking Center and one in Cyfronet in Cracow. On these servers all the computations will be performed. They are connected with a very fast fiber channel, as shown in Figure 7. Users can access the application through the portal located at website of the PROGRESS Project [20].

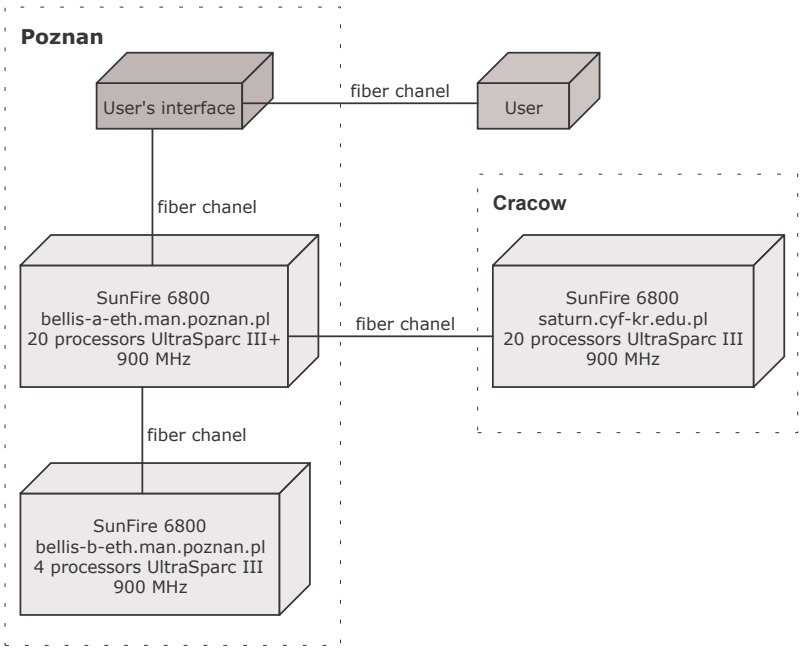

Figure 7. The system physical layout.

3.1.2. Outline of the system operation

The communication between processes is ensured by an MPI library (namely, MPICH-G2 [30]). The distribution model is based on the master-slave architecture,

therefore two kinds of processes can be distinguished – the master (one) and the slaves (one or more).

The master process supervises the computations. Among its tasks are:

- loading the input data,
- sending the data to the slave processes,
- distributing tasks to the slave processes,
- receiving the results from the slave processes,
- measuring the computation time,
- finishing the computations,
- presenting the overall results of the computations.

A slave process (or in general, the set of slave processes) has the following tasks:

- receiving the data from the master process,
- processing the received data,
- sending the result of computations to the master process.

3.1.3. General idea of implementation

One of the key implementation requirements was simplification of a further development. For that reason, the communication code has been separated from the computational one. As a result, changes in computational parts of the algorithms do not require changing the model of communication between processes. Moreover, there is an option to compile the code into both the sequential (single-process) and the distributed version. This is implemented with the use of a preprocessor flag – `MPI`, as presented in the following example.

```
#ifdef MPI
   /* source code only for distributed version */
#endif
   /* common source code for both versions */
#ifndef MPI
   /* source code only for sequential version */
#endif
```

In the distributed version the computational parts of the sequential code are executed mainly by the slave processes. The master process only (with some exceptions) manages the communication and the distribution of tasks to the slave processes.

As mentioned in the sections 2.1 and 2.2, the algorithm consists of four stages, executed one after another. Here, the stages will be further divided into smaller pieces (referred to as parts, or distributed parts in case of ambiguity), which are performed in a

sequence, one after another – an output of one part is an input of the next one. Since a way of distribution of every part is more or less the same, a kind of general template is presented here (on the example of arc reduction stage, which consists only of one part). All parts are based on this template, with some slight modifications, explained in sections 3.2 and 3.3.

```
ArcReduction(parameters...)
{
#ifdef MPI                  // distributed version
   if (nMpiMyId == 0)       // master process
   {
      /* send the data to the slave processes */
      while (!all_the_computations_are_done)
      {
         /* send data range to a free slave processes
            (or wait if there are no free slaves) */
      }
      /* inform slaves that the part is finished */
      /* receive partial results from slave processes
         and consolidate them */
   }
   else                     // slave process
   {
      AR_Receiver(parameters...)
   }
#endif
#ifndef MPI                 // sequential version
   foreach (vertex in graph)
   {
      AR_OneRow(vertex, other_parameters...)
   }
#endif
}

AR_Receiver(parameters...)
{
   /* receive data from master process */
   while (!part_is_finished)
   {
      /* receive range to compute */
      foreach (vertex in range)
      {
         AR_OneRow(vertex, other_parameters...)
      }
   }
   /* send local results to master process */
}

AR_OneRow(vertex, other_parameters...)
{
   /* do the sequential part of algorithm for the vertex */
}
```

The template requires a few words of comments. If only it is necessary, the master process sends the data to the slaves, which is the input of the part. Then, the master distributes tasks among the slaves, by sending them ranges of data to compute. A range is

in fact two numbers, denoting an interval of sequences, vertices or arcs to serve (for the arc reduction example above, a range defines a set of vertices, for which outgoing arcs are to be checked and determined whether they should be deleted). The size of a range (called also a pack size) has been established experimentally, to achieve optimal efficiency. When a slave receives a range, it performs calculations for all the sequences/vertices/arcs from the range (according to the appropriate sequential algorithm presented in the section 2). When it finishes, it reports that fact to the master, and waits for a new assignment. When the master process notices, that a slave has finished its job, thus is free, it sends a next range of data to compute to the slave (even if other slaves are still performing their tasks), unless there is no data left. Finally, when all the tasks are distributed, the master awaits for all the slaves to finish. Then, it informs all the slaves about finishing of the part and collects the results sent by them. Collected results are consolidated and broadcasted to slaves if necessary.

3.2. Algorithm for the ideal case

3.2.1. Overlap graph construction

Before starting this stage the master process sends all input sequences to the slave processes.

In the first step of this stage the window size is calculated in the same way as in the sequential algorithm, by each slave independently (the calculated value is identical for all of them). After that, slaves locally create a hash table with characteristic numbers for the beginning (i.e. the first window) of each input sequence.

Next step is a calculation of all possible successors for each sequence in the constructed graph. This is realized as a distributed part according to the previously described template. The slaves, when getting a range, compute successors in the constructed graph (i.e. overlapping sequences – see section 2.1.1 for details), for all sequences in the range. The pack size is set to 100.

During slaves' computations, apart from finding successors, they constantly check whether a sequence is a subsequence of another one. The slaves send list of newly computed subsequences to the master. Then, the master updates its array of subsequences and sends this information to other slaves (along with next data ranges). Thanks to that, each process may update its local array of subsequences and later use it for pruning of computations. It considerably shortens the computation time.

The schema in Figure 8 presents the way the master process communicates with the slave processes in this stage. To improve readability only one slave process is shown. The communication with the other processes is identical.

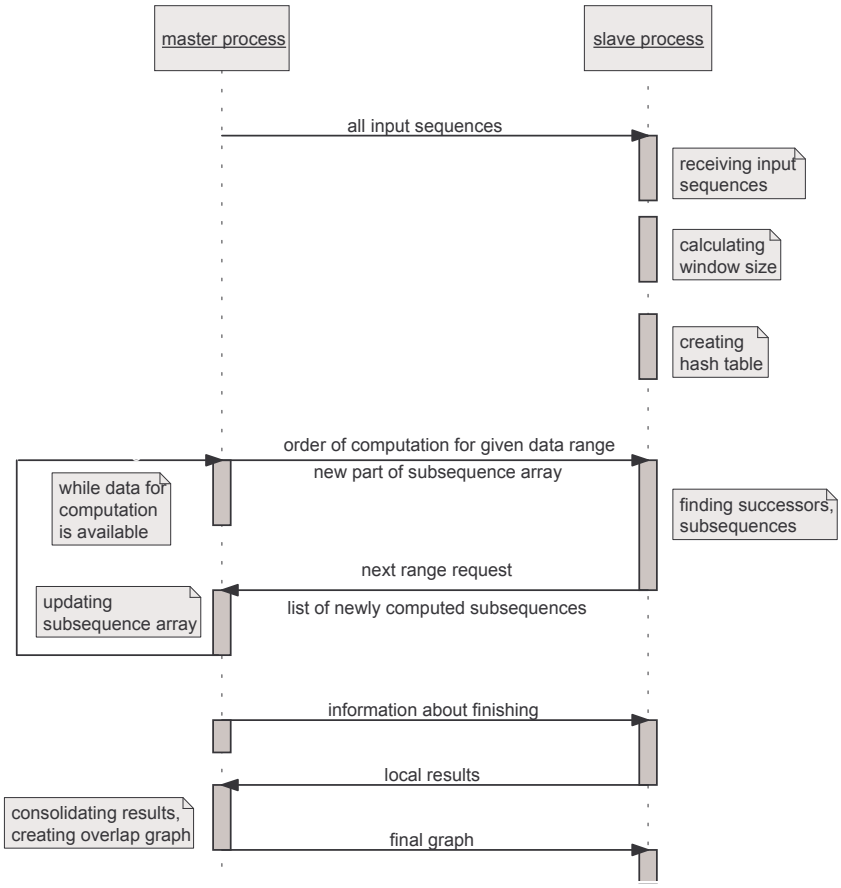

Figure 8. Schema of distribution of the overlap graph creation (ideal case).

3.2.2. Arc reduction

As already mentioned, this stage is a distributed part itself. Again, general template of distribution is applied to computations. This time, when a slave receives a range, it looks for arcs to delete, which are outgoing from all the vertices in the range (this is done exactly as in the sequential version, described in the section 2.1.2). The pack size is set to 50.

A detailed schema of distribution in the arc reduction stage is presented in Figure 9.

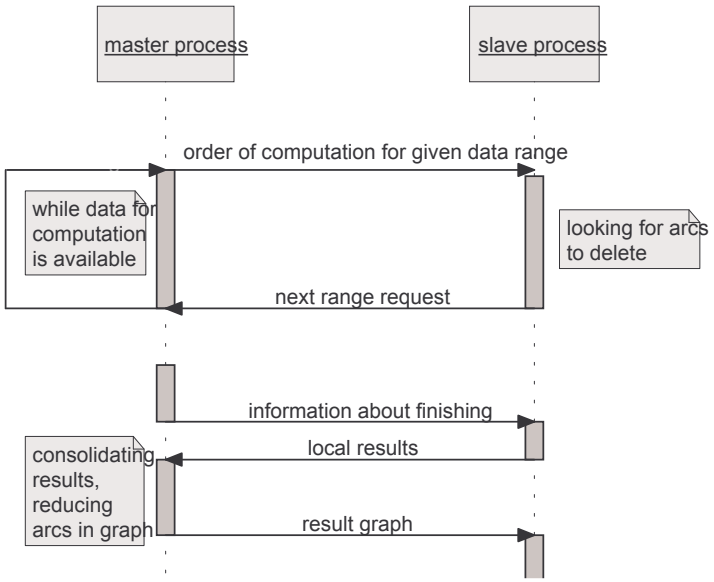

Figure 9. Schema of distribution of the arc reduction stage (ideal case).

3.2.3. Finding path

In this stage, the way of distribution is somewhat different than in the previous ones, but the general template is still used. Finding the first element of a path and reordering of fragments are done sequentially by the master (just as described in the sections 2.1.3.1 and 2.1.3.3), since they are executed very rarely. Only finding next elements of a path might be parallelized, provided that the number of successors of the last element of a path is big enough. Actually, a process of selection of a single next element of a path is a distributed part by itself (if only parallelization is used).

Having already a part of the path, the master distributes tasks of checking the arcs outgoing from the last vertex of the path, according to the general template (the pack size is set to 10). When getting a range, a slave computes the value of the *f* function for each arc in that range (just as in the section 2.1.3.2). When all the arcs are served, the slaves send the best value of *f* found by them (together with an arc, the value has been found for), the master aggregates the results and chooses the next element of the path – now this part is considered to be finished and the next one is started.

However, it very often happens that the number of outgoing arcs from the last vertex of the path is very small. In such a case, a time cost of communication between processes would be greater than a time cost of local computation on one processor. So, there is a threshold introduced (set experimentally to 2,000), under which a next vertex is searched sequentially in the master. In fact, this threshold is hardly ever reached, so usually the most efficient way is to compute it sequentially.

One may notice that if all vertices have low number of successors, then this stage is not parallelized at all. There is nothing to worry about. This stage is extremely fast even if performed sequentially (especially, when the vertices do not have many successors). The tests have proven that computation times of this stage are only a tiny part of the computation times of the entire application.

A detailed schema of distribution of this stage is presented in Figure 10.

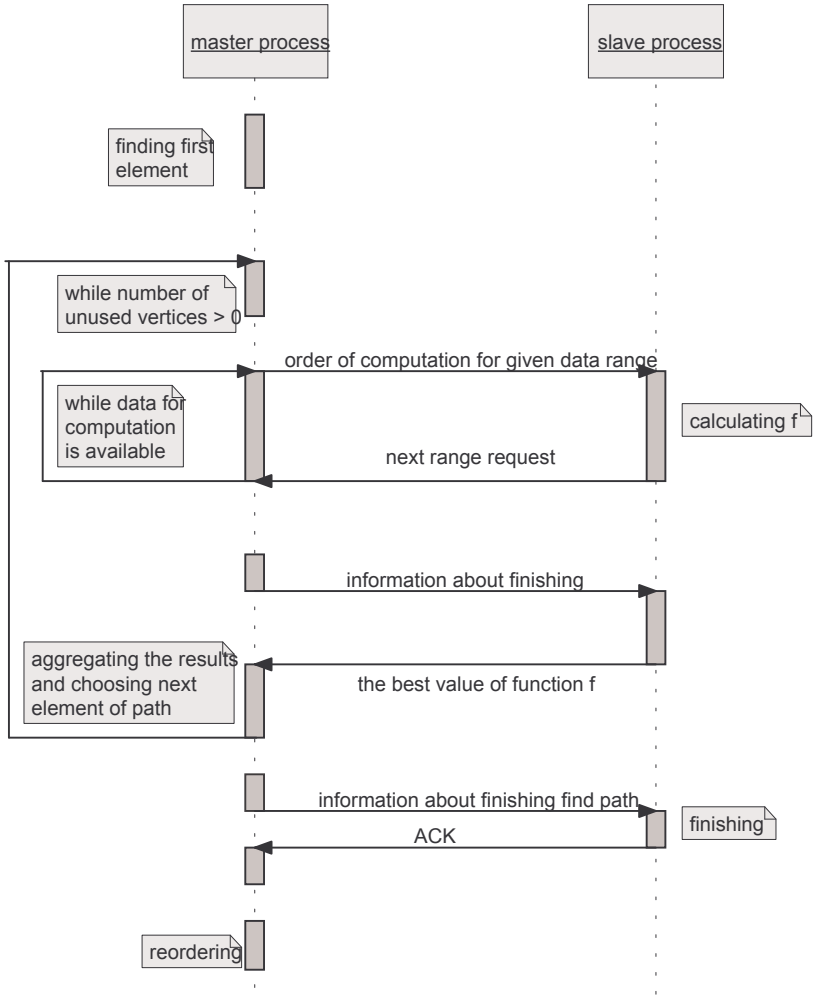

Figure 10. Schema of distribution of stage of finding path (ideal case).

3.2.4. Output sequence construction

This stage is straightforward and computationally easy, so it is performed sequentially by the master.

3.3. Algorithm for the real case

3.3.1. Overlap graph construction

As in the ideal case, before starting this stage the master process sends all input sequences to the slave processes. After that, the window size is computed in the same way as in the sequential algorithm (see the section 2.2.2). Each process computes this number individually, but obviously its value is identical for all of them.

Next, a hash table is created (in a parallel way), which includes characteristic numbers for every window of every sequence. Again, the described template of distribution is used. After getting a range, a slave computes characteristic numbers for all windows of all sequences in the range. The pack size is set to 1. At the end of this part, when master receives the results, they are merged into one hash table, which is sent to all the slaves.

The next, and the most important action of this stage, is the computation of successors for each vertex in the constructed overlap graph. This is done exactly in the same way as it was described in the section 3.2.1, but of course this time slaves perform a part of the sequential algorithm for the real case (see the section 2.2.2) instead of the ideal case. The pack size is set to 10.

See Figure 11 for detailed distribution schema.

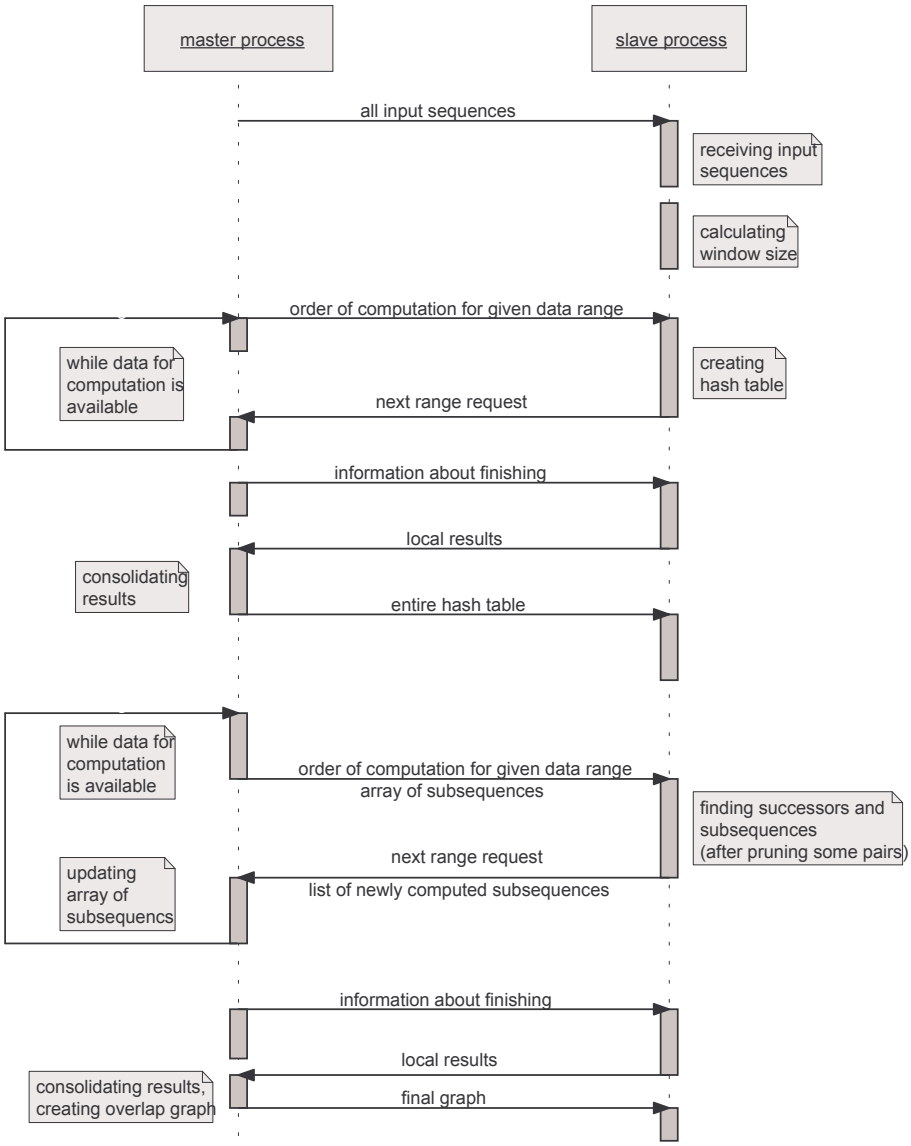

Figure 11. Schema of distribution of the overlap graph creation (real case).

3.3.2. Arc reduction

As mentioned in the section 2.2.3, this stage is identical to the corresponding stage in the ideal case version of the algorithm. Thus, of course, schema of distribution does not change. Only the pack size is changed, due to larger number of arcs it has to deal with. This time it is set to 10.

3.3.3. Finding path

This stage slightly differs from the corresponding stage of the ideal case version (as explained in the section 2.2.4). However, the difference is very little, so the schema of distribution is not affected at all.

3.3.4. Output sequence construction

Although this stage is much more complex than its counterpart in the algorithm for the ideal case, it is also feasible to be performed sequentially by the master.

4. Results

To examine the correctness, computational complexity and parallelization efficiency of ASM, a series of tests were applied. The input instances were mainly artificially prepared basing on sequences coming from chromosome arm 2R of Drosophila Melanogaster genome and 21st chromosome of the human genome. The already assembled sequences were multiplied and randomly cut to have lengths from a few dozens to 1,000 bp (500 bp in average), to simulate a shotgun process. Then, random 20% of pieces were dropped and, in case of the real case algorithm, the rest were randomly (uniformly) stuffed with insertion, deletion and substitution errors. However, the ultimate correctness and efficiency test was done on the genome of an RNA coronavirus causing SARS (SARS-CoV), for which the input sequences come directly from a shotgun sequencing process. The shotgun data were obtained from Canada's Michael Smith Genome Sciences Centre [22].

The subsection 4.1 presents the efficiency of ASM, in both the sequential (single processor conducting all the computations) and the distributed (master process distributing tasks to slaves) version. The tests for the ideal case were performed on both the Drosophila Melanogaster and the human data. In fact, the algorithm is so fast, that it spends a significant amount of time only on the latter data, which are huge. The real case algorithm is not that fast so it was tested only on part of the Drosophila Melanogaster data. The correctness results are presented in the subsection 4.2. The tests were performed only on the part of the Drosophila Melanogaster data. The subsection 4.3 describes the results achieved on the SARS-CoV data. Obviously, here the ideal case was not tested, because of presence of errors in the real data. The last subsection (4.4) summarizes and interprets the results.

4.1. Efficiency tests

4.1.1. Algorithm for the ideal case

The tests for the ideal case were conducted with the *mo* parameter set to 10. The results presented in Figure 12 show the computation time of sequential version for the input data not greater than 16 MB (Drosophila Melanogaster data). The time complexity of the algorithm seems to be linear with regard to the instance size, which is a great result. Unfortunately, for larger instances (human data), one can observe time complexity

slightly over linear (see Figure 13). However, it still allows to assembly very large instance – hundreds of megabytes can be processed in quite a short time.

The results presented in Figures 13 and 14 show, that the distributed version is quite scalable. These experiments were performed with the sequential and the distributed version using 2, 4, 8 and 12 slaves. Figure 13 presents the assembly time on many processors and contrasts it to the sequential version. The use of many processors allowed to significantly shorten the time of computations. This makes possible to assemble longer genomes in a short time. Figure 14 shows the efficiency of parallel computations being the ratio of the computation time of the sequential version to the time of the distributed one, the latter multiplied by the number of slaves (the master can work on one processor with a slave without affecting its performance, thus it is not included in the formula). Although the sequential algorithm is very fast, a good efficiency was obtained. The efficiency is far from the optimal (which is 1) for large number of slaves, because of the time complexity of the algorithm. It is close to linear, so reading the data from a file and sending it to all the processes, which cannot be parallelized, takes a great part of the execution time.

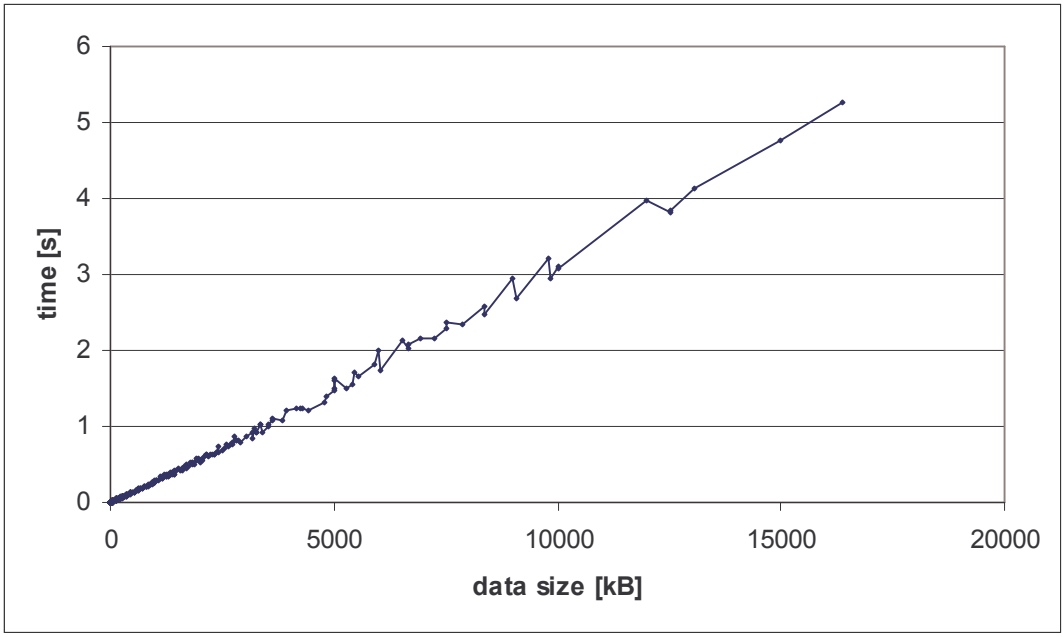

Figure 12. Computation time of sequential version of the algorithm for the ideal case.

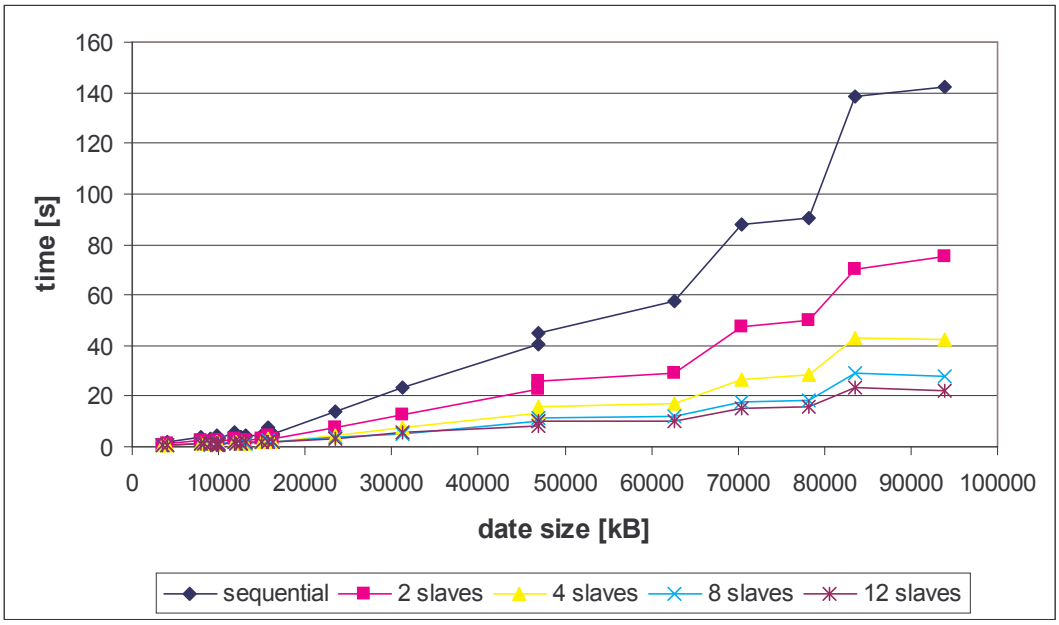

Figure 13. Computation time of sequential and distributed versions of the algorithm for ideal case.

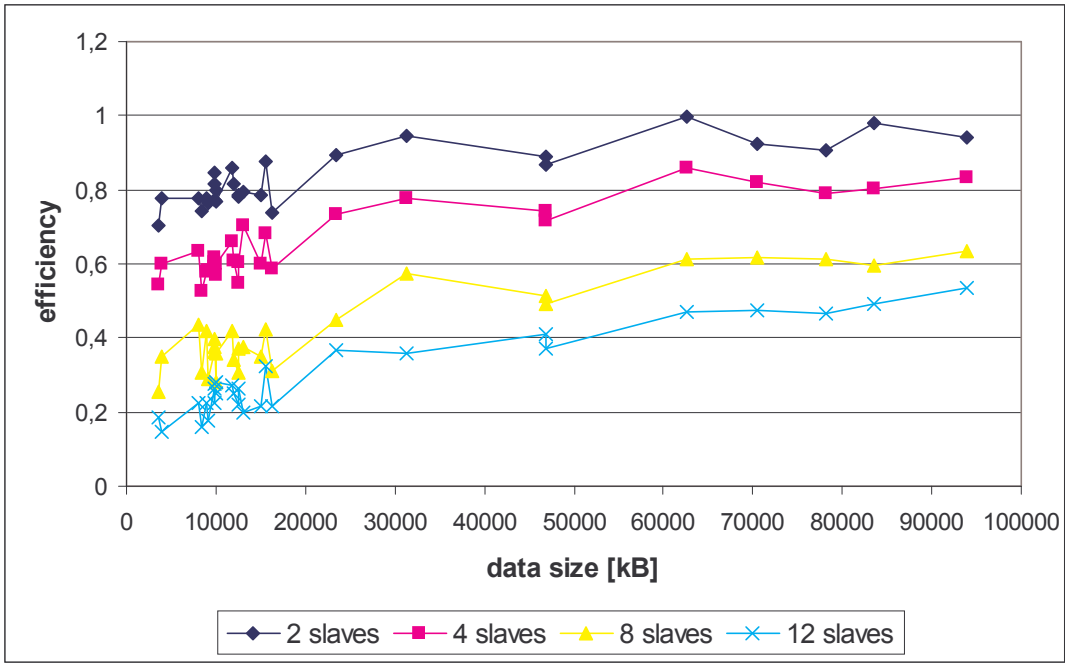

Figure 14. Efficiency of the distributed computations of the algorithm for the ideal case.

4.1.2. Algorithm for the real case

The tests for the real case were performed with the following values of parameters: *eb* = 0.1 and *mo* = 10. Figures 15 and 16 show that time complexity of the algorithm is far from being linear (with regard to the instance size). However, it is not square as well – it looks like something between (note that by doubling the size of the input instance, time of computations grows about three times).

Figures 16 and 17 show the results of the algorithm parallelization. Figure 17 presents the efficiency of computations in the distributed environment in comparison to the sequential algorithm (the configuration is the same as stated in the section 4.1.1). Excellent results, allowing processing of much longer sequences in a short time, were obtained. The efficiency of computations (determined as in the section 4.1.1) is very high. For large instances it is about 90% or better, also for a large number of processors (see Figure 17).

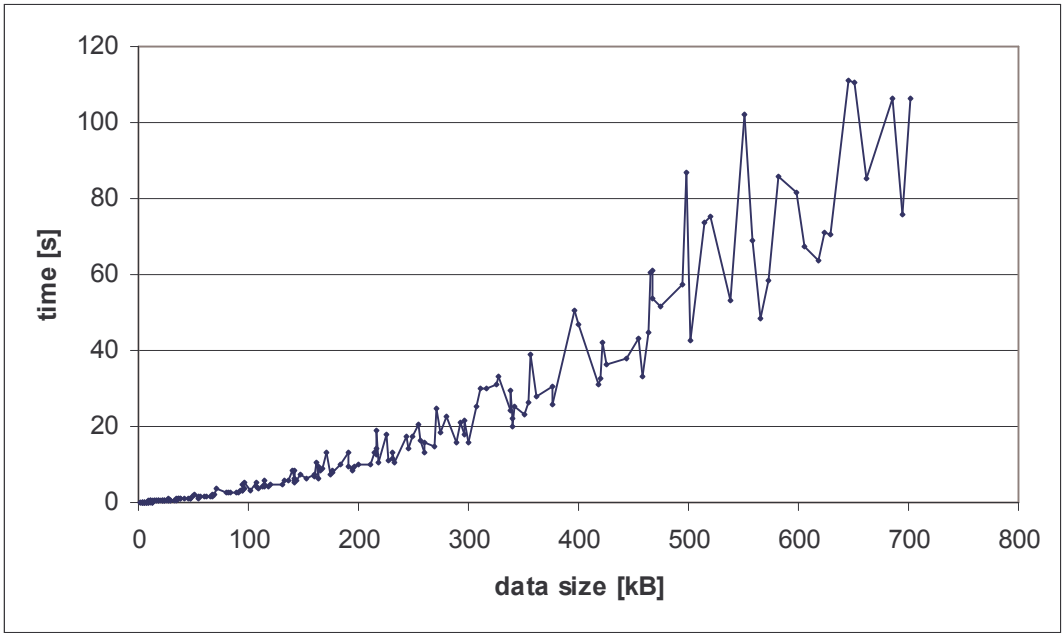

Figure 15. Computation time of sequential version of the algorithm for the real case.

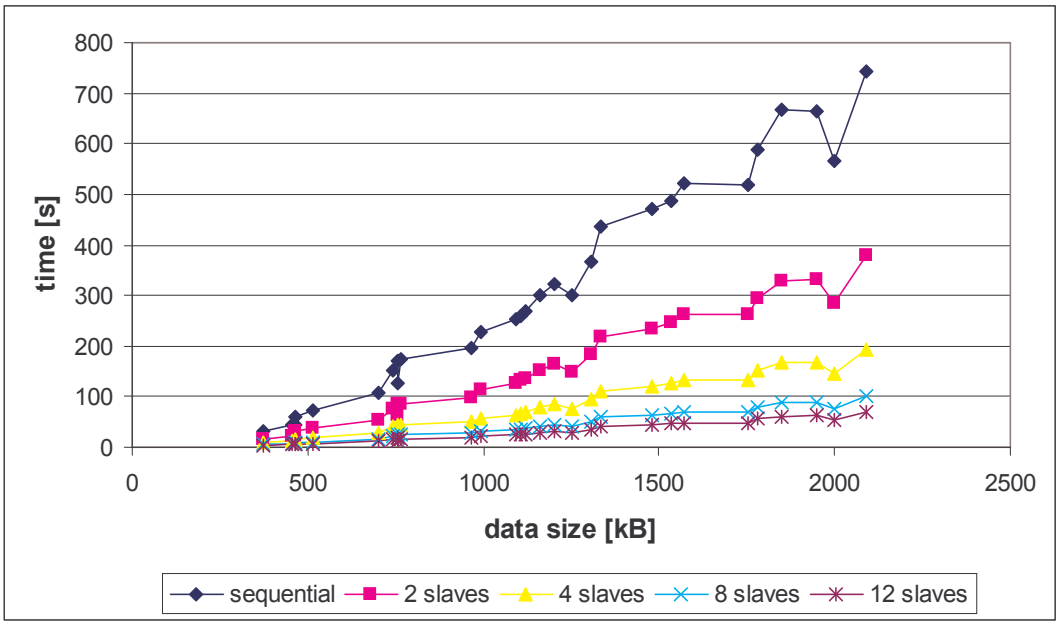

Figure 16. Computation time of sequential and distributed versions of the algorithm for the real case.

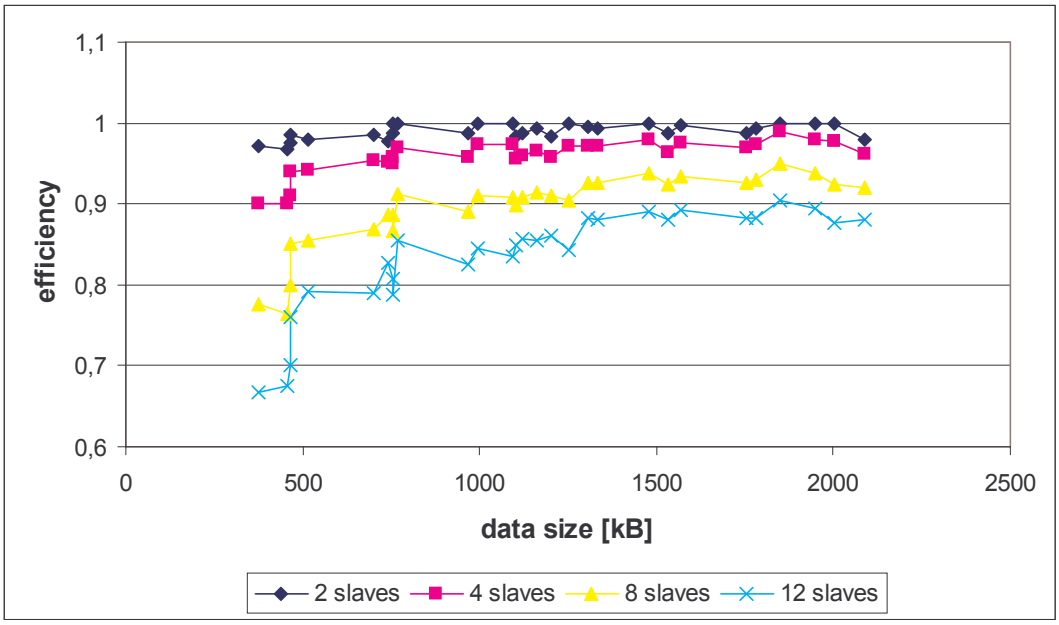

Figure 17. Efficiency of the distributed computations of the algorithm for the real case.

4.2. Correctness tests

Before the results are presented, a few additional words about the test instances should be said, in order to properly understand the tables with the results. There were 85 sequences of different sizes chosen from the Drosophila Melanogaster chromosome, labeled with names from `seq#1` to `seq#85` (see Table 1). To crate input instances for the ideal case, they were multiplied 6, 8 or 10 times and then prepared as explained in the beginning of the section 4. The names of input instances are created according to the example `seq#24x8.ex`, which means that it was obtained from `seq#24`, multiplied 8 times and have no errors ("ex" stands for "exact"). Input instances for the real case are created in the same way, but additionally there are inaccuracy errors introduced at some positions with probability of 2%. The rule of the name creation is the same, but extension `.inex` is used instead of `.ex` (which stands for "inexact").

Name	Size [bp]
seq#24	519
seq#8	1103
seq#73	1137
seq#7	1248
seq#18	1626
seq#44	1766
seq#35	2487
seq#55	3437
seq#23	3609
seq#19	4191
seq#15	4343
seq#2	5889
seq#75	6267

Name	Size [bp]
seq#79	37121
seq#33	37733
seq#28	42748
seq#30	45289
seq#59	46065
seq#14	46103
seq#43	53010
seq#41	54066
seq#1	54103
seq#50	58586
seq#47	71600
seq#49	73696
seq#27	79091

Name	Size [bp]
seq#64	234518
seq#38	236769
seq#60	242623
seq#42	258015
seq#39	279870
seq#16	286862
seq#54	295500
seq#53	310524
seq#37	325490
seq#67	334110
seq#20	334979
seq#22	347828
seq#78	362727

seq#4	7553	seq#84	89212	seq#77	453807
seq#11	8447	seq#71	97107	seq#31	490011
seq#10	11289	seq#13	100607	seq#82	540240
seq#57	12766	seq#74	100924	seq#81	551420
seq#56	14575	seq#51	106854	seq#40	615871
seq#69	15019	seq#70	116368	seq#58	709973
seq#9	19769	seq#52	125769	seq#65	852949
seq#17	22449	seq#85	128098	seq#68	853855
seq#6	24450	seq#61	146926	seq#66	926932
seq#5	26002	seq#76	149374	seq#72	1007350
seq#32	28720	seq#45	159713	seq#48	1280440
seq#12	30151	seq#46	163200	seq#36	1282578
seq#34	30470	seq#25	165939	seq#80	1531061
seq#3	30867	seq#62	173529	seq#83	1671491
seq#21	34176	seq#26	207546		
seq#63	35063	seq#29	208403		

Table 1. Sizes of original sequences for correctness tests.

4.2.1. Algorithm for the ideal case

Table 2 presents the results for the ideal case, with the *mo* parameter set to 10. The first column of the table is an instance name (as just explained). The second column is the number of sequences not contained in each other, produced by ASM (1 is the most preferable). The last column denotes percent correctness of the longest of the generated sequences comparing to the original sequence. In case this sequence does not cover the entire genome, its percent length (comparing to the length of the original sequence) is shown in brackets.

Name	#	Align. [%]	Name	#	Align. [%]	Name	#	Align. [%]
seq#24x6.ex	1	100.00	seq#24x8.ex	1	100.00	seq#24x10.ex	1	100.00
seq#8x6.ex	1	100.00	seq#8x8.ex	1	100.00	seq#8x10.ex	1	100.00
seq#73x6.ex	1	100.00	seq#73x8.ex	1	100.00	seq#73x10.ex	1	100.00
seq#7x6.ex	1	100.00	seq#7x8.ex	1	100.00	seq#7x10.ex	1	100.00
seq#18x6.ex	1	100.00	seq#18x8.ex	1	100.00	seq#18x10.ex	1	100.00
seq#44x6.ex	1	100.00	seq#44x8.ex	1	100.00	seq#44x10.ex	1	100.00
seq#35x6.ex	1	100.00	seq#35x8.ex	1	100.00	seq#35x10.ex	1	100.00
seq#55x6.ex	1	100.00	seq#55x8.ex	1	100.00	seq#55x10.ex	1	100.00
seq#23x6.ex	1	100.00	seq#23x8.ex	1	100.00	seq#23x10.ex	1	100.00
seq#19x6.ex	1	100.00	seq#19x8.ex	1	100.00	seq#19x10.ex	1	100.00
seq#15x6.ex	1	100.00	seq#15x8.ex	1	100.00	seq#15x10.ex	1	100.00
seq#2x6.ex	1	100.00	seq#2x8.ex	1	100.00	seq#2x10.ex	1	100.00
seq#75x6.ex	1	100.00	seq#75x8.ex	1	100.00	seq#75x10.ex	1	100.00
seq#4x6.ex	1	100.00	seq#4x8.ex	1	100.00	seq#4x10.ex	1	100.00
seq#11x6.ex	1	100.00	seq#11x8.ex	1	100.00	seq#11x10.ex	1	100.00
seq#10x6.ex	1	100.00	seq#10x8.ex	1	100.00	seq#10x10.ex	1	100.00
seq#57x6.ex	1	100.00	seq#57x8.ex	1	100.00	seq#57x10.ex	1	100.00
seq#56x6.ex	1	100.00	seq#56x8.ex	1	100.00	seq#56x10.ex	1	100.00
seq#69x6.ex	1	100.00	seq#69x8.ex	1	100.00	seq#69x10.ex	1	100.00
seq#9x6.ex	1	100.00	seq#9x8.ex	1	100.00	seq#9x10.ex	1	100.00
seq#17x6.ex	2	100.00 (95.08)	seq#17x8.ex	1	100.00	seq#17x10.ex	1	100.00
seq#6x6.ex	1	100.00	seq#6x8.ex	1	100.00	seq#6x10.ex	1	100.00

seq#5x6.ex	1	100.00	seq#5x8.ex	1	100.00	seq#5x10.ex	1	100.00
seq#32x6.ex	1	100.00	seq#32x8.ex	1	100.00	seq#32x10.ex	1	100.00
seq#12x6.ex	1	100.00	seq#12x8.ex	1	100.00	seq#12x10.ex	1	100.00
seq#34x6.ex	1	100.00	seq#34x8.ex	1	100.00	seq#34x10.ex	1	100.00
seq#3x6.ex	1	100.00	seq#3x8.ex	1	100.00	seq#3x10.ex	1	100.00
seq#21x6.ex	2	100.00 (85.73)	seq#21x8.ex	1	100.00	seq#21x10.ex	1	100.00
seq#63x6.ex	1	100.00	seq#63x8.ex	1	100.00	seq#63x10.ex	1	100.00
seq#79x6.ex	1	100.00	seq#79x8.ex	1	100.00	seq#79x10.ex	1	100.00
seq#33x6.ex	1	100.00	seq#33x8.ex	1	100.00	seq#33x10.ex	1	100.00
seq#28x6.ex	1	100.00	seq#28x8.ex	1	100.00	seq#28x10.ex	1	100.00
seq#30x6.ex	1	100.00	seq#30x8.ex	1	100.00	seq#30x10.ex	1	100.00
seq#59x6.ex	3	100.00 (72.08)	seq#59x8.ex	1	100.00	seq#59x10.ex	1	100.00
seq#14x6.ex	1	100.00	seq#14x8.ex	1	100.00	seq#14x10.ex	1	100.00
seq#43x6.ex	1	100.00	seq#43x8.ex	1	100.00	seq#43x10.ex	1	100.00
seq#41x6.ex	1	100.00	seq#41x8.ex	1	100.00	seq#41x10.ex	1	100.00
seq#1x6.ex	1	100.00	seq#1x8.ex	1	100.00	seq#1x10.ex	1	100.00
seq#50x6.ex	3	100.00 (94.16)	seq#50x8.ex	1	100.00	seq#50x10.ex	1	100.00
seq#47x6.ex	1	100.00	seq#47x8.ex	1	100.00	seq#47x10.ex	1	100.00
seq#49x6.ex	2	100.00 (86.93)	seq#49x8.ex	1	100.00	seq#49x10.ex	1	100.00
seq#27x6.ex	1	100.00	seq#27x8.ex	1	100.00	seq#27x10.ex	1	100.00
seq#84x6.ex	3	100.00 (61.76)	seq#84x8.ex	1	100.00	seq#84x10.ex	1	100.00
seq#71x6.ex	1	100.00	seq#71x8.ex	1	100.00	seq#71x10.ex	1	100.00
seq#13x6.ex	1	100.00	seq#13x8.ex	1	100.00	seq#13x10.ex	1	100.00
seq#74x6.ex	1	100.00	seq#74x8.ex	1	100.00	seq#74x10.ex	1	100.00
seq#51x6.ex	3	100.00 (69.84)	seq#51x8.ex	1	100.00	seq#51x10.ex	1	100.00
seq#70x6.ex	1	100.00	seq#70x8.ex	1	100.00	seq#70x10.ex	1	100.00
seq#52x6.ex	1	100.00	seq#52x8.ex	1	100.00	seq#52x10.ex	1	100.00
seq#85x6.ex	1	100.00	seq#85x8.ex	1	100.00	seq#85x10.ex	1	100.00
seq#61x6.ex	2	100.00 (99.77)	seq#61x8.ex	1	100.00	seq#61x10.ex	1	100.00
seq#76x6.ex	1	100.00	seq#76x8.ex	1	100.00	seq#76x10.ex	1	100.00
seq#45x6.ex	2	100.00 (97.64)	seq#45x8.ex	2	100.00 (97.64)	seq#45x10.ex	2	100.00 (60.41)
seq#46x6.ex	4	100.00 (54.70)	seq#46x8.ex	1	100.00	seq#46x10.ex	1	100.00
seq#25x6.ex	2	100.00 (59.50)	seq#25x8.ex	1	100.00	seq#25x10.ex	1	100.00
seq#62x6.ex	2	100.00 (90.70)	seq#62x8.ex	1	100.00	seq#62x10.ex	1	100.00
seq#26x6.ex	1	100.00	seq#26x8.ex	1	100.00	seq#26x10.ex	1	100.00
seq#29x6.ex	2	100.00 (57.45)	seq#29x8.ex	1	100.00	seq#29x10.ex	1	100.00
seq#64x6.ex	2	100.00 (48.39)	seq#64x8.ex	1	100.00	seq#64x10.ex	1	100.00
seq#38x6.ex	3	100.00 (76.41)	seq#38x8.ex	1	100.00	seq#38x10.ex	1	100.00
seq#60x6.ex	1	100.00	seq#60x8.ex	1	100.00	seq#60x10.ex	1	100.00
seq#42x6.ex	2	100.00 (68.13)	seq#42x8.ex	1	100.00	seq#42x10.ex	1	100.00
seq#39x6.ex	2	100.00 (92.45)	seq#39x8.ex	1	100.00	seq#39x10.ex	1	100.00
seq#16x6.ex	3	100.00 (62.02)	seq#16x8.ex	2	100.00 (97.47)	seq#16x10.ex	1	100.00

seq#54x6.ex	3	100.00 (52.95)	seq#54x8.ex	1	100.00	seq#54x10.ex	1	100.00
seq#53x6.ex	2	100.00 (88.73)	seq#53x8.ex	1	100.00	seq#53x10.ex	1	100.00
seq#37x6.ex	2	100.00 (63.48)	seq#37x8.ex	1	100.00	seq#37x10.ex	1	100.00
seq#67x6.ex	2	100.00 (83.77)	seq#67x8.ex	1	100.00	seq#67x10.ex	1	100.00
seq#20x6.ex	2	100.00 (69.30)	seq#20x8.ex	1	100.00	seq#20x10.ex	1	100.00
seq#22x6.ex	2	100.00 (82.81)	seq#22x8.ex	1	100.00	seq#22x10.ex	1	100.00
seq#78x6.ex	2	100.00 (68.46)	seq#78x8.ex	1	100.00	seq#78x10.ex	1	100.00
seq#77x6.ex	6	100.00 (36.30)	seq#77x8.ex	1	100.00	seq#77x10.ex	1	100.00
seq#31x6.ex	2	100.00 (55.32)	seq#31x8.ex	1	100.00	seq#31x10.ex	1	100.00
seq#82x6.ex	2	100.00 (65.37)	seq#82x8.ex	2	100.00 (82.15)	seq#82x10.ex	1	100.00
seq#81x6.ex	3	100.00 (57.33)	seq#81x8.ex	1	100.00	seq#81x10.ex	1	100.00
seq#40x6.ex	6	100.00 (32.20)	seq#40x8.ex	1	100.00	seq#40x10.ex	1	100.00
seq#58x6.ex	1	100.00	seq#58x8.ex	1	100.00	seq#58x10.ex	1	100.00
seq#65x6.ex	5	100.00 (33.78)	seq#65x8.ex	1	100.00	seq#65x10.ex	1	100.00
seq#68x6.ex	7	100.00 (52.39)	seq#68x8.ex	1	100.00	seq#68x10.ex	1	100.00
seq#66x6.ex	7	100.00 (31.54)	seq#66x8.ex	1	100.00	seq#66x10.ex	1	100.00
seq#72x6.ex	5	100.00 (41.02)	seq#72x8.ex	2	100.00 (96.62)	seq#72x10.ex	1	100.00
seq#48x6.ex	7	100.00 (52.42)	seq#48x8.ex	1	100.00	seq#48x10.ex	1	100.00
seq#36x6.ex	4	100.00 (45.72)	seq#36x8.ex	1	100.00	seq#36x10.ex	1	100.00
seq#80x6.ex	5	100.00 (49.01)	seq#80x8.ex	3	100.00 (38.68)	seq#80x10.ex	2	100.00 (67.56)
seq#83x6.ex	7	100.00 (27.93)	seq#83x8.ex	5	100.00 (30.12)	seq#83x10.ex	5	100.00 (27.93)

Table 2. Results of the correctness tests for the ideal case.

As one can see the results are perfect in almost every case, as long as appropriate multiplication is provided. Only multiplication of 6 seems to have problems with producing one connected contig for greater sequences (thus indicating the lower bound of data quality that ASM can deal with). However, it is not a surprise, since 6 is widely regarded as a minimal reasonable multiplication and deletion of 20% of sequences makes things even worse. Additionally, although it is not presented in the table (due to lack of wood in the universe), in most cases, the remaining, smaller contigs also match in 100% and altogether they cover almost the entire original sequence – this causes the results not to be that bad as they might have appeared.

4.2.2. Algorithm for the real case

Table 3 presents the results for the real case, with the *mo* parameter set to 10 and *eb* set to 0.08. The columns of the table have exactly the same meaning as in the previous section.

Name	#	Align. [%]	Name	#	Align. [%]	Name	#	Align. [%]
seq#24x6.inex	1	97.78	seq#24x8.inex	1	97.30	seq#24x10.inex	1	98.26
seq#8x6.inex	1	98.46	seq#8x8.inex	1	98.73	seq#8x10.inex	1	98.55
seq#73x6.inex	1	98.24	seq#73x8.inex	1	97.80	seq#73x10.inex	1	98.59
seq#7x6.inex	1	97.71	seq#7x8.inex	1	98.88	seq#7x10.inex	1	98.68
seq#18x6.inex	1	97.54	seq#18x8.inex	1	98.65	seq#18x10.inex	1	98.46
seq#44x6.inex	1	94.84	seq#44x8.inex	2	97.90 (94.39)	seq#44x10.inex	1	97.51
seq#35x6.inex	1	97.55	seq#35x8.inex	1	97.30	seq#35x10.inex	1	98.71
seq#55x6.inex	1	97.82	seq#55x8.inex	1	97.82	seq#55x10.inex	1	98.34
seq#23x6.inex	1	97.70	seq#23x8.inex	1	96.80	seq#23x10.inex	1	98.13
seq#19x6.inex	1	97.94	seq#19x8.inex	1	98.70	seq#19x10.inex	1	98.25
seq#15x6.inex	1	97.87	seq#15x8.inex	1	96.59	seq#15x10.inex	1	96.41
seq#2x6.inex	1	97.56	seq#2x8.inex	1	98.40	seq#2x10.inex	1	98.50
seq#75x6.inex	1	98.17	seq#75x8.inex	2	97.87 (56.65)	seq#75x10.inex	1	98.36
seq#4x6.inex	1	98.24	seq#4x8.inex	1	98.32	seq#4x10.inex	1	98.19
seq#11x6.inex	1	98.13	seq#11x8.inex	1	98.24	seq#11x10.inex	1	98.26
seq#10x6.inex	2	98.35 (82.07)	seq#10x8.inex	2	97.59 (90.99)	seq#10x10.inex	1	98.17
seq#57x6.inex	1	97.95	seq#57x8.inex	1	97.50	seq#57x10.inex	1	98.19
seq#56x6.inex	1	98.09	seq#56x8.inex	1	96.86	seq#56x10.inex	1	98.16
seq#69x6.inex	1	97.82	seq#69x8.inex	1	98.00	seq#69x10.inex	1	98.13
seq#9x6.inex	2	98.12 (69.55)	seq#9x8.inex	1	98.22	seq#9x10.inex	1	97.75
seq#17x6.inex	1	98.00 (95.08)	seq#17x8.inex	1	97.92	seq#17x10.inex	2	98.14 (86.31)
seq#6x6.inex	1	98.06	seq#6x8.inex	1	98.34	seq#6x10.inex	1	98.29
seq#5x6.inex	1	97.91	seq#5x8.inex	1	97.98	seq#5x10.inex	1	97.97
seq#32x6.inex	1	98.33	seq#32x8.inex	1	97.84	seq#32x10.inex	1	98.11
seq#12x6.inex	1	98.09	seq#12x8.inex	1	97.95	seq#12x10.inex	1	98.02
seq#34x6.inex	1	97.64	seq#34x8.inex	1	97.97	seq#34x10.inex	1	98.11
seq#3x6.inex	1	97.99	seq#3x8.inex	1	97.93	seq#3x10.inex	1	97.43
seq#21x6.inex	2	98.20 (85.73)	seq#21x8.inex	2	97.91 (62.39)	seq#21x10.inex	1	98.33
seq#63x6.inex	1	97.87 (96.75)	seq#63x8.inex	1	98.29	seq#63x10.inex	1	98.09
seq#79x6.inex	3	98.16 (52.08)	seq#79x8.inex	1	97.83	seq#79x10.inex	1	98.30
seq#33x6.inex	2	98.09 (60.27)	seq#33x8.inex	2	98.13 (58.23)	seq#33x10.inex	1	98.04
seq#28x6.inex	1	98.07	seq#28x8.inex	1	98.20	seq#28x10.inex	1	98.34
seq#30x6.inex	1	97.97	seq#30x8.inex	1	98.30	seq#30x10.inex	1	98.23
seq#59x6.inex	2	98.22 (72.06)	seq#59x8.inex	1	98.09	seq#59x10.inex	1	98.08
seq#14x6.inex	2	97.86 (63.73)	seq#14x8.inex	2	98.24 (75.27)	seq#14x10.inex	1	98.18
seq#43x6.inex	3	97.83 (44.55)	seq#43x8.inex	3	98.08 (47.57)	seq#43x10.inex	1	97.95
seq#41x6.inex	3	98.06 (57.85)	seq#41x8.inex	2	94.65 (55.70)	seq#41x10.inex	2	74.80 (55.35)
seq#1x6.inex	2	98.20 (95.49)	seq#1x8.inex	2	98.25 (73.65)	seq#1x10.inex	1	98.22
seq#50x6.inex	2	98.17 (94.16)	seq#50x8.inex	1	98.16	seq#50x10.inex	1	98.15

seq#47x6.inex	1	97.81	seq#47x8.inex	1	98.09	seq#47x10.inex	1	98.24
seq#49x6.inex	2	97.79 (86.93)	seq#49x8.inex	2	98.09 (71.38)	seq#49x10.inex	1	98.15
seq#27x6.inex	2	97.98 (64.61)	seq#27x8.inex	1	98.17	seq#27x10.inex	1	98.37
seq#84x6.inex	4	79.73 (43.99)	seq#84x8.inex	3	96.56 (78.12)	seq#84x10.inex	3	98.18 (69.97)
seq#71x6.inex	2	97.94 (90.5)	seq#71x8.inex	1	98.25	seq#71x10.inex	1	98.20
seq#13x6.inex	1	98.10	seq#13x8.inex	2	97.91 (57.52)	seq#13x10.inex	1	97.88
seq#74x6.inex	2	98.07 (93.13)	seq#74x8.inex	1	98.24	seq#74x10.inex	1	97.99
seq#51x6.inex	2	98.10 (69.84)	seq#51x8.inex	2	98.27 (67.56)	seq#51x10.inex	1	98.06
seq#70x6.inex	2	98.12 (59.12)	seq#70x8.inex	1	98.18	seq#70x10.inex	1	97.95
seq#52x6.inex	3	98.02 (59.42)	seq#52x8.inex	2	98.12 (63.75)	seq#52x10.inex	1	97.98
seq#85x6.inex	2	98.04 (75.95)	seq#85x8.inex	1	98.11 (95.52)	seq#85x10.inex	2	98.13 (87.82)
seq#61x6.inex	1	98.04	seq#61x8.inex	1	98.20	seq#61x10.inex	1	97.96
seq#76x6.inex	4	97.95 (42.89)	seq#76x8.inex	1	98.04	seq#76x10.inex	1	98.05
seq#45x6.inex	2	98.12 (56.73)	seq#45x8.inex	1	94.43 (95.71)	seq#45x10.inex	2	97.81 (60.41)
seq#46x6.inex	4	98.11 (38.87)	seq#46x8.inex	2	97.96 (52.59)	seq#46x10.inex	3	98.24 (49.12)
seq#25x6.inex	5	98.10 (36.99)	seq#25x8.inex	1	98.09	seq#25x10.inex	1	98.17
seq#62x6.inex	3	97.83 (50.44)	seq#62x8.inex	1	86.14	seq#62x10.inex	1	97.74
seq#26x6.inex	1	97.99 (97.31)	seq#26x8.inex	1	98.18 (98.01)	seq#26x10.inex	2	98.23 (91.66)
seq#29x6.inex	4	97.99 (42.48)	seq#29x8.inex	4	98.11 (44.24)	seq#29x10.inex	1	98.02
seq#64x6.inex	5	98.05 (37.63)	seq#64x8.inex	2	97.52 (67.33)	seq#64x10.inex	1	98.09
seq#38x6.inex	5	98.18 (39.21)	seq#38x8.inex	2	98.14 (57.52)	seq#38x10.inex	1	98.07
seq#60x6.inex	1	97.89	seq#60x8.inex	1	98.06	seq#60x10.inex	3	98.10 (58.67)
seq#42x6.inex	5	97.83 (29.85)	seq#42x8.inex	2	98.11 (89.71)	seq#42x10.inex	1	98.12
seq#39x6.inex	5	97.91 (50.60)	seq#39x8.inex	2	97.98 (55.92)	seq#39x10.inex	2	98.21 (65.81)
seq#16x6.inex	5	83.92 (38.45)	seq#16x8.inex	4	98.03 (37.94)	seq#16x10.inex	3	98.04 (50.01)
seq#54x6.inex	5	97.88 (50.25)	seq#54x8.inex	1	98.09	seq#54x10.inex	1	98.00
seq#53x6.inex	4	98.00 (40.30)	seq#53x8.inex	2	97.98 (50.29)	seq#53x10.inex	1	97.93
seq#37x6.inex	3	97.88 (56.90)	seq#37x8.inex	2	98.04 (90.49)	seq#37x10.inex	1	98.02
seq#67x6.inex	5	97.90 (25.41)	seq#67x8.inex	2	98.17 (65.22)	seq#67x10.inex	2	96.13 (82.27)
seq#20x6.inex	5	97.97 (40.32)	seq#20x8.inex	2	98.11 (80.44)	seq#20x10.inex	2	98.08 (84.48)
seq#22x6.inex	4	98.08 (47.48)	seq#22x8.inex	3	98.28 (45.74)	seq#22x10.inex	1	98.21
seq#78x6.inex	8	97.98 (22.35)	seq#78x8.inex	3	97.90 (50.71)	seq#78x10.inex	3	98.15 (45.27)
seq#77x6.inex	7	98.02 (27.83)	seq#77x8.inex	4	98.22 (52.35)	seq#77x10.inex	2	98.08 (67.61)
seq#31x6.inex	4	98.03	seq#31x8.inex	3	98.03	seq#31x10.inex	1	97.98

		(44.68)			(52.16)			
seq#82x6.inex	7	96.99 (20.31)	seq#82x8.inex	5	74.47 (54.84)	seq#82x10.inex	2	98.10 (55.56)
seq#81x6.inex	6	98.08 (42.93)	seq#81x8.inex	4	98.09 (38.87)	seq#81x10.inex	3	98.19 (59.00)
seq#40x6.inex	8	97.92 (13.44)	seq#40x8.inex	3	98.11 (46.04)	seq#40x10.inex	3	97.96 (66.48)
seq#58x6.inex	7	97.78 (17.95)	seq#58x8.inex	4	98.04 (37.41)	seq#58x10.inex	4	96.62 (52.28)
seq#65x6.inex	9	97.91 (11.39)	seq#65x8.inex	6	98.13 (26.80)	seq#65x10.inex	2	98.06 (68.14)
seq#68x6.inex	9	98.06 (12.31)	seq#68x8.inex	4	97.98 (37.51)	seq#68x10.inex	3	98.10 (41.30)
seq#66x6.inex	8	98.16 (13.51)	seq#66x8.inex	2	98.04 (82.50)	seq#66x10.inex	2	98.10 (91.02)
seq#72x6.inex	8	98.05 (15.10)	seq#72x8.inex	5	98.08 (33.47)	seq#72x10.inex	3	97.99 (36.01)
seq#48x6.inex	9	98.07 (9.96)	seq#48x8.inex	4	98.05 (50.13)	seq#48x10.inex	4	96.94 (33.56)
seq#36x6.inex	6	98.02 (28.53)	seq#36x8.inex	4	98.04 (38.80)	seq#36x10.inex	3	98.07 (58.98)
seq#80x6.inex	8	98.09 (12.27)	seq#80x8.inex	8	97.94 (23.30)	seq#80x10.inex	4	98.01 (42.72)
seq#83x6.inex	5	97.98 (9.50)	seq#83x8.inex	9	98.05 (17.93)	seq#83x10.inex	4	98.16 (27.93)

Table 3. Results of the correctness tests for the real case.

The results are also quite good. In almost vast majority of cases the quality (messered by means of alignment with the original sequences) of the largest contig is about 98%, as one might expected. In almost all cases it is greater than 96.5%. As far as number of contigs is concerned, the results are visibly worse than in the ideal case. Quite often ASM does not manage to produce a single connected contig. One can observe a threshold of length of sequences, over which the application almost always produces several contings. Fortunatelly, for good multiplication the threshold is high – 300.000 bp for multiplication of 10 and 200.000 bp for 8 (multiplication of 6 cannot be regarded as good in presence of errors and with 20% of sequences deleted). Again, as in the ideal case, for many instances the quality of excessive contings (not presented in the table) is very good and altogether they cover almost the entire original sequence. This allows us to think, that overall results are satisfactory, proving that the presented application may be successfully used in practice.

4.2.3. Two-strand version

As explained in the section 2.3, the algorithm for each sequence produces its reverse complement. As the number of sequences is doubled, there is a greater chance that some sequences would be erroneously interpreted as overlapping. Therefore, the results are slightly worse. A drop in quality for 3% of instances was observed for the ideal case and 5% for the real case. However, the results for these instances were not that bad. Although

ASM did not produce one sequence, it managed to reconstruct the original sequence in few parts of quite a good quality.

To work around the problem, greater values of *mo* are recommended.

4.3. Tests on SARS-CoV

The file downloaded from Canada's Michael Smith Genome Sciences Centre contains sequences coming from seven clones, referred to as SARS11, SARS12, SARS211, SARS212, SARS213, SARS214 and SARS215. At the beginning, SARS11 and SARS12 were excluded from further consideration, due to small size of these sets and high rate of unknown nucleotides (~45% of N letters in the sequences). The SARS215 set has the best quality of data and thus, it was mainly focused on while assembling the SARS-CoV genome. However, the rest of the clones were helpful in the finishing part and were also used in efficiency testing of the presented method.

Since the obtained sequences were not quality clipped, they required to be cut. After some analysis, the sequences from SARS215 were trimmed by 90 nucleotides at the beginning and 40 at the end. Sequences from the other sets were trimmed by 150 nucleotides at the beginning and 650 at the end, because of their low quality.

4.3.1. The obtained sequence

To achieve the best possible result, various combinations of the algorithm parameters were used, but $eb = 0.04$ and $mo = 40$ turned out to be the best in this case. Of course, *ts* parameter had to be used, as the input sequences were coming from both DNA strands.

For the above parameters, the ASM application found five main contigs, partially overlapping with each other, which lengths range from about 2,500 bp to about 9,500 bp. The contigs cover 99.8% of the entire genome. The results were compared to the sequence assembled by Canada's Michael Smith Genome Sciences Centre (also available in NCBI [31] with the name AY274119.3). The comparison proved that these five contigs are subsequences of AY274119.3, as expected. Moreover, they match (by means of alignment) in about 99%. See Table 4 for details.

Align. with AY274119.3	% length of AY274119.3	Position in AY274119.3
99.49%	31.88%	17-9501
99.17%	23.98%	16213-23346
99.20%	22.24%	23121-29738
99.29%	18.95%	11167-16805
98.34%	8.80%	8976-11593

Table 4. Comparison of contigs produced by ASM with AY274119.3.

The obtained result is very good and allowed for easy manual finishing of the assembly process. The manually constructed genome is almost identical to AY274119.3 sequence. There are only two small differences between them:

- nucleotide 10,249 is `U` instead of `C`;
- instead of `AUAUUAGGUUUU` at the beginning of the sequence, there is `AAUUCGCGGCCGCGUCG`.

However, due to low coverage of these areas, it is impossible to determine which results are correct. It requires additional research. Anyway, these results prove that the sequence assembled by Canada's Michael Smith Genome Sciences Centre is correct (with regard to small variations) and at the same time verify the presented approach in practice.

4.3.2. Comparison with other assembly applications

To further evaluate the quality of the result and also the efficiency of ASM, it was compared to two other, assembly applications – Phrap [14] and CAP3 [17]. Both were run with the same trimmed SARS215 sequences on input, as described above. Phrap produced three overlapping contigs, one about 18,500 bp and two about 6,000 bp long, covering also about 99.8% of the entire genome. The first two match AY274119.3 in about 98.2% and the third one in 97.2%, which is 1-2% quality drop comparing to ASM. CAP3 produced four contigs of lengths from 1,000 bp to 11,000 pb, covering barely 98% of the genome. Three of them almost perfectly match AY274119.3, but the fourth in about 98.5%. See Tables 5 and 6 for details and contrast them to Table 4 in the previous section.

Align. with AY274119.3	% length of AY274119.3	Position in AY274119.3
98.10%	62.26%	11213-29734
98.29%	20.30%	5555-11593
97.17%	20.21%	33-6045

Table 5. Comparison of contigs produced by Phrap with AY274119.3.

Align. with AY274119.3	% length of AY274119.3	Position in AY274119.3
99.94%	49.17%	13380-28007
98.32%	37.75%	17-11246
99.82%	9.30%	10956-13722
99.41%	4.01%	28550-29742

Table 6. Comparison of contigs produced by CAP3 with AY274119.3.

As far as time efficiency is concerned, all the three methods were tested on SARS211, SARS212, SARS213, SARS214 and SARS215 sets. For this particular test, in order to make equal chances for all the applications, the parallel computation abilities of ASM

were disabled. However, note that the ability to be executed in a distributed system is a great advantage of ASM. The computation times are presented in Table 7.

Clone	Input size	CAP3	Phrap	ASM
SARS211	371 kB	236 min	7 s	217 s
SARS212	275 kB	52 min	8 s	66 s
SARS213	244 kB	35 min	8 s	57 s
SARS214	152 kB	23 min	3 s	25 s
SARS215	489 kB	61 min	23 s	176 s

Table 7. Execution times of CAP3, Phrap and ASM for SARS-CoV data.

One can see that the best results were obtained using ASM, thus simplifying the job of scientists to the necessary minimum. CAP3 has also very good results (contigs usually are slightly better than in case of ASM), but the price is high – computation time is very long and not all the genome is covered. Only Phrap was able to produce few very long contigs, covering almost the entire genome like ASM and the computation time is quite impressive. However, quality of the contigs was not the best. To summarize, all the methods are quite good and each one surpass others on a particular aspect. All produce the results of quality good enough to easily do the finishing. However, only ASM managed to cover almost perfectly the entire genome in a short time, thus proving its usability.

4.4. Summary

The design of good algorithms for the overlap graph construction and the distributed implementation of assembly algorithms led us to excellent results. The computation time is very short for the sequential algorithm and can be further significantly shortened when the application is executed in a parallel way. Especially parallelization of the algorithm for the real case, which is more computationally expensive, has a very high efficiency, enabling to apply the computational power of the distributed systems in order to drastically shorten the time of assembling. Although such a good efficiency does not appear in the algorithm for the ideal case, the sequential algorithm itself is extremely fast and parallelization gives a visible improvement. This all allows assembling very long genomes, which makes the application greatly useful in practice.

As far as correctness is concerned, the application was tested on a large variety of instances and managed to perform very well. Although, the results were not always perfect, they were still very good and the final test on SARS-CoV genome was passed with an excellent mark.

To sum up, a good algorithmic design, together with a distributed environment provided for the project, resulted in creation of a very fast and useful application, which gives good quality results. The application can successfully compete with other, widely used assembly software.

REFERENCES

[1] Burland V, Plunkett G 3rd, Sofia HJ, Daniels DL, Blattner FR. (1995); *Analysis of the Escherichia coli genome VI: DNA sequence of the region from 92.8 through 100 minutes*. Nucleic Acids Res.; **23:** 2105-2119.

[2] Goffeau A, Barrell BG, Bussey H, Davis RW, Dujon B, Feldmann H, Galibert F, Hoheisel JD, Jacq C, Johnston M, Louis EJ, Mewes HW, Murakami Y, Philippsen P, Tettelin H, Oliver SG. (1996); *Life with 6000 genes*. Science; **274:** 546-567.

[3] Press release, *Celera Genomics Completes the First Assembly of the Human Genome*, http://www.celera.com/celera/pr_1056581295.

[4] The Human Genome Organisation, http://www.gene.ucl.ac.uk/hugo/.

[5] Publication, AHFMR Magazine, *Human Genome Organization (HUGO)*, http://www.ahfmr.ab.ca/publications/newsletter/SeptOct98/genome.shtml.

[6] Los Alamos National Laboratory, Bioscience Division, http://bioscience.lanl.gov/.

[7] Bains W. (1991); *Hybridization methods for DNA sequencing*. Genomics; **11:** 294-301.

[8] Setubal J, Meidanis J. (1997); *Introduction to Computational Molecular Biology*. PWS Publishing Company, Boston.

[9] Błażewicz J, Formanowicz P, Kasprzak M, Markiewicz WT, Węglarz J. (1999); *DNA sequencing with positive and negative errors*. J. Comp. Biol.; **6:** 113-123.

[10] Błażewicz J, Formanowicz P, Kasprzak M, Markiewicz WT, Węglarz J. (2000); *Tabu search for DNA sequencing with false negatives and false positives*. European Journal of Operational Research; **125:** 257-265.

[11] Venter C (et al). (2001); *The Sequence of the Human Genome*. Science; **291:** 1304-1351.

[12] European Bioinformatics Institute, http://www.ebi.ac.uk/.

[13] University of Washington Genome Center, http://www.genome.washington.edu/.

[14] The Phred/Phrap/Consed System, http://www.phrap.org/.

[15] The Institute for Genomic Research, http://www.tigr.org/.

[16] Biotechnology Computing Facility – FAKtory Online Resources, http://bcf.arl.arizona.edu/faktory/.

[17] CAP3 Sequence Assembly Program, http://genome.cs.mtu.edu/cap/cap3.html.

[18] CAP4 – Paracel's DNA Sequence Assembly Program, http://ludwig-sun2.unil.ch/~gtomio/est/cap4.pdf.

[19] Staden Package, http://staden.sourceforge.net/.

[20] PROGRESS, http://progress.psnc.pl/.

[21] Kasprzak M. (2002); *ASSEMBL – opis programu i wyniki obliczeń na procesorze Intel Pentium 4* (in Polish). Report RA-002/2002; Poznań Supercomputing and Networking Center.

[22] Canada's Michael Smith Genome Sciences Centre, SARS-associated Coronavirus, http://www.bcgsc.ca/bioinfo/SARS/.

[23] Marra MA (et al). (2003); *The Genome Sequence of the SARS-Associated Coronavirus*. Science; **300:** 1339-1404.

[24] Cormen TH, Leiserson CE, Rivest RL. (1990); *Introduction to Algorithms*. MIT Press.

[25] Gutin G, Punnen AP (Eds.). (2002); *Travelling Salesman Problem and its Variations*. Kluwer Academic Publishers, Dordrecht.

[26] Waterman MS. (1995); *Introduction to Computational Biology. Maps, Sequences and Genomes*. Chapman & Hall, London.

[27] Pevzner PA. (2000); *Computational Molecular Biology. An Algorithmic Approach*. MIT Press, Cambridge, London.

[28] Smith TF, Waterman MS. (1981); *Identification of common molecular subsequences*. J. Mol. Biol.; **147:** 195-7.

[29] Wilbur WJ, Lipman DJ. (1983); *Rapid similarity searches of nucleic acid and protein data banks*. Proc Natl Acad Sci USA; **80:** 726-730.

[30] MPICH-G2, http://www3.niu.edu/mpi/.

[31] National Center for Biotechnology Information, http://www.ncbi.nih.gov/.