Assembling DNA Fragments with a Distributed Genetic Algorithm

Gabriel Luque and **Enrique Alba**
Universidad de Málaga, Complejo Tecnológico
Campus de Teatinos, 29071 Málaga, Spain
{gabriel,eat}@lcc.uma.es

Sami Khuri
Department of Computer Science
San José State University
One Washington Square
San José, CA 95192-0249
khuri@cs.sjsu.edu

December 9, 2004

Abstract

As more research centers embark on sequencing new genomes, the problem of DNA fragment assembly for shotgun sequencing is growing in importance and complexity. Accurate and fast assembly is a crucial part of any sequencing project and many algorithms have been developed to tackle it. Since the DNA fragment assembly problem is NP-hard, exact solutions are very difficult to obtain. Various heuristics, including genetic algorithms, were designed for solving the fragment assembly problem. While the sequential genetic algorithm has given good results, it is unable to sequence very large DNA molecules. In this work, we present a distributed genetic algorithm that surmounts that problem. We show how the distributed genetic algorithm can tackle problem instances that are 77K base pairs long accurately.

Keywords: DNA Fragment Assembly Problem, Genetic Algorithms, Distributed Genetic Algorithms.

1 Introduction

DNA fragment assembly is a technique that attempts to reconstruct the original DNA sequence from a large number of fragments, each one having several hundred base-pairs (bps) long. The DNA fragment assembly is needed because current technology, such as gel electrophoresis, cannot directly and accurately

sequence DNA molecules longer than 1000 bases. However, most genomes are much longer. For example, a human DNA is about 3.2 billion nucleotides in length and cannot be read at once.

The following technique was developed to deal with this limitation. First, the DNA molecule is amplified, that is, many copies of the molecule are created. The molecules are then cut at random sites to obtain fragments that are short enough to be sequenced directly. The overlapping fragments are then assembled back into the original DNA molecule. This strategy is called *shotgun sequencing*. Originally, the assembly of short fragments was done by hand, which is inefficient and error-prone. Hence, a lot of effort has been put into finding techniques to automate the shotgun sequence assembly. Over the past decade a number of fragment assembly packages have been developed and used to sequence different organisms. The most popular packages are PHRAP [7], TIGR assembler [18], STROLL [4], CAP3 [9], Celera assembler [12], and EULER [16]. These packages deal with the previously described challenges to different extend, but none of them solves them all. Each package automates fragment assembly using a variety of algorithms. The most popular techniques are greedy-based. This work reports on the design and implementation of a parallel distributed genetic algorithm to tackle the DNA fragment assembly problem.

The remainder of this chapter is organized as follows. In the next section, we present background information about the DNA fragment assembly problem. In Section 3, the details of the sequential Genetic Algorithm (GA) are presented. We discuss the GA operators, fitness functions [14], and how to design and implement a GA for the DNA fragment assembly problem. In Section 4, we present the parallel distributed GA. We analyze the results of our experiments in Section 5. We end this chapter by giving our final thoughts and conclusions in Section 6.

2 The DNA Fragment Assembly Problem

We start this section by giving a vivid analogy to the fragment assembly problem: "Imagine several copies of a book cut by scissors into thousands of pieces, say 10 millions. Each copy is cut in an individual way such that a piece from one copy may overlap a piece from another copy. Assume one million pieces lost and remaining nine million are splashed with ink, try to recover the original text." [16]. We can think of the DNA target sequence as being the original text and the DNA fragments are the pieces cut out from the book. To further understand the problem, we need to know the following basic terminology:

- **Fragment:** A short sequence of DNA with length up to 1000 bps.
- **Shotgun data:** A set of fragments.
- **Prefix:** A substring comprising the first n characters of fragment f.
- **Suffix:** A substring comprising the last n characters of fragment f.

- **Overlap:** Common sequence between the suffix of one fragment and the prefix of another fragment.
- **Layout:** An alignment of collection of fragments based on the overlap order.
- **Contig:** A layout consisting of contiguous overlapping fragments.
- **Consensus:** A sequence derived from the layout by taking the majority vote for each column of the layout.

To measure the quality of a consensus, we can look at the distribution of the coverage. Coverage at a base position is defined as the number of fragments at that position. It is a measure of the redundancy of the fragment data. It denotes the number of fragments, on average, in which a given nucleotide in the target DNA is expected to appear. It is computed as the number of bases read from fragments over the length of the target DNA [17].

$$Coverage = \frac{\sum_{i=1}^{n} length\ of\ the\ fragment\ i}{target\ sequence\ length} \tag{1}$$

where n is the number of fragments. TIGR uses the coverage metric to ensure the correctness of the assembly result. The coverage usually ranges from 6 to 10 [10]. The higher the coverage, the fewer the gaps are expected, and the better the result.

2.1 DNA Sequencing Process

To determine the function of specific genes, scientists have learned to read the sequence of nucleotides comprising a DNA sequence in a process called DNA sequencing. The fragment assembly starts with breaking the given DNA sequence into small fragments. To do that, multiple exact copies of the original DNA sequence are made. Each copy is then cut into short fragments at random positions. These are the first three steps depicted in Figure 1 and they take place in the laboratory. After the fragment set is obtained, traditional assemble approach is followed in this order: overlap, layout, and then consensus. To ensure that enough fragments overlap, the reading of fragments continues until the coverage is satisfied. These steps are the last three steps in Figure 1. In what follows, we give a brief description of each of the three phases, namely overlap, layout, and consensus.

Overlap Phase - Finding the overlapping fragments.
This phase consists in finding the best or longest match between the suffix of one sequence and the prefix of another. In this step, we compare all possible pairs of fragments to determine their similarity. Usually, the dynamic programming algorithm applied to semiglobal alignment is used in this step. The intuition behind finding the pairwise overlap is that fragments with a significant overlap

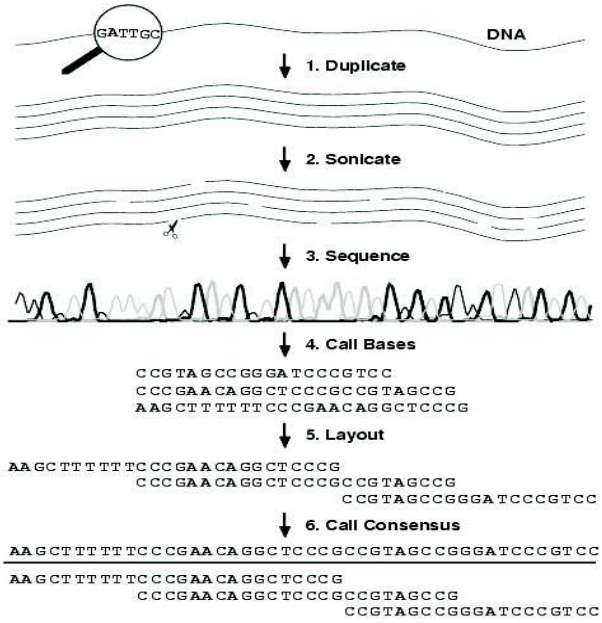

Figure 1: Graphical representation of DNA sequencing and assembly [3]

score are very likely next to each other in the target sequence.

Layout Phase - Finding the order of fragments based on the computed similarity score. This is the most difficult step because it is hard to tell the true overlap due to the following challenges:

1. Unknown orientation: After the original sequence is cut into many fragments, the orientation is lost. The sequence can be read in either 5' to 3' or 3' to 5'. One does not know which strand should be selected. If one fragment does not have any overlap with another, it is still possible that its reverse complement might have such an overlap.

2. Base call errors: There are three types of base call errors: substitution, insertion, and deletion errors. They occur due to experimental errors in the electrophoresis procedure. Errors affect the detection of fragment overlaps. Hence, the consensus determination requires multiple alignments in high coverage regions.

3. Incomplete coverage: It happens when the algorithm is not able to assemble a given set of fragments into a single contig.

4. Repeated regions: Repeats are sequences that appear two or more times in the target DNA. Repeated regions have caused problems in many genome-

sequencing projects, and none of the current assembly programs can handle them perfectly.

5. Chimeras and contamination: Chimeras arise when two fragments that are not adjacent or overlapping on the target molecule join together into one fragment. Contamination occurs due to the incomplete purification of the fragment from the vector DNA.

After the order is determined, the progressive alignment algorithm is applied to combine all the pairwise alignments obtained in the overlap phase.

Consensus Phase - Deriving the DNA sequence from the layout. The most common technique used in this phase is to apply the majority rule in building the consensus.

Example: We next give an example of the fragment assembly process.

Given a set of fragments {`F1 = GTCAG, F2 = TCGGA, F3 = ATGTC, F4 = CGGATG`}, assume the four fragments are read from 5' to 3' direction. First, we need to determine the overlap of each pair of the fragments by the using semiglobal alignment algorithm. Next, we determine the order of the fragments based on the overlap scores, which are calculated in the overlap phase. Suppose we have the following order: `F2 F4 F3 F1`. Then, the layout and the consensus for this example can be constructed as follows:

```
    F2 ->    TCGGA
    F4 ->     CGGATG
    F3 ->        ATGTC
    F1 ->          GTCAG
-------------------------
Consensus -> TCGGATGTCAG
```

In this example, the resulting order allows to build a sequence having just one contig. Since finding the exact order takes a huge amount of time, a heuristic such as Genetic Algorithm can be applied in this step [13, 14, 15]. In the following section, we illustrate how the Genetic Algorithm is implemented for the DNA fragment assembly problem.

3 DNA Fragment Assembly Using the Sequential GA

The Genetic Algorithm (GA) was invented in the mid-1970s by John Holland [8]. It is based on Darwin's Evolution Theory. GA uses the concept of survival of the fittest and natural selection to evolve a population of individuals over many generations by using different operators: selection, crossover, and mutation. As the generations are passed along, the average fitness of the population is likely to improve. Genetic Algorithm can be used for optimization problems with

multiple parameters and multiple objectives. It is commonly used to tackle NP-hard problems such as the DNA fragment assembly and the Travelling Salesman Problem (TSP). NP-hard problems require tremendous computational resources to solve exactly. Genetic Algorithms help to find good solutions in a reasonable amount of time. Next, we present the sequential GA for the fragment assembly problem. More details about the inner workings of the algorithm can be found in [11].

1. Randomly generate the initial population of fragment orderings.
2. Evaluate the population by computing fitness.
3. while(NOT termination condition)
   (a) Select fragment orderings for the next generation through ranking selection
   (b) Alter population by
      i. applying the crossover operator
      ii. applying the mutation operator
      iii. re-evaluate the population.

3.1 Implementation Details

Let us give some details about the most important issues of our implementation.

Population Representation

We use the permutation representation with integer number encoding. A permutation of integers represents a sequence of fragment numbers, where successive fragments overlap. The population in this representation requires a list of fragments assigned with a unique integer ID. For example, 8 fragments will need eight identifiers: 0, 1, 2, 3, 4, 5, 6, 7. The permutation representation requires special operators to make sure that we always get legal (feasible) solutions. In order to maintain a legal solution, the two conditions that must be satisfied are (1) all fragments must be presented in the ordering, and (2) no duplicate fragments are allowed in the ordering. For example, one possible ordering for 4 fragments is 3 0 2 1. It means that fragment 3 is at the first position and fragment 0 is at the second position, and so on.

Population Size

We use a fixed size population to initialize random permutations.

Program Termination

The program can be terminated in one of two ways. We can specify the maximum number of generations to stop the algorithm or we can also stop the algorithm when the solution is no longer improving.

Fitness Function

A fitness function is used to evaluate how good a particular solution is. It is applied to each individual in the population and it should guide the genetic algorithm towards the optimal solution. In the DNA fragment assembly problem, the fitness function measures the multiple sequences alignment quality and finds the best scoring alignment. Parsons, Forrest, and Burks mentioned two different fitness functions [14].

Fitness function $F1$ - sums the overlap score for adjacent fragments in a given solution. When this fitness function is used, the objective is to maximize such a score. It means that the best individual will have the highest score.

$$F1(l) = \sum_{i=0}^{n-2} w \cdot (f[i]f[i+1]) \tag{2}$$

Fitness function $F2$ - not only sums the overlap score for adjacent fragments, but it also sums the overlap score for all other possible pairs.

$$F2(l) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} |i-j| \times w \cdot (f[i]f[j]) \tag{3}$$

This fitness function penalizes solutions in which strong overlaps occur between non-adjacent fragments in the layouts. When this fitness function is used, the objective is to minimize the overlap score. It means that the best individual will have the lowest score.

The overlap score in both $F1$ and $F2$ is computed using the semiglobal alignment algorithm.

Recombination Operator

Two or more parents are recombined to produce one or more offspring. The purpose of this operator is to allow partial solutions to evolve in different individuals and then combine them to produce a better solution. It is implemented by running through the population and for each individual, deciding whether it should be selected for crossover using a parameter called *crossover rate* (P_c). A crossover rate of 1.0 indicates that all the selected individuals are used in the crossover. Thus, there are no survivors. However, empirical studies have shown that better results are achieved by a crossover rate between 0.65 and 0.85, which implies that the probability of an individual moving unchanged to the next generation ranges from 0.15 to 0.35.

For our experimental runs, we use the order-based crossover (OX) and the edge-recombination crossover (ER). These operator were specifically designed for tackling problems with permutation representations.

The order-based crossover operator first copies the fragment ID between two random positions in Parent1 into the offspring's corresponding positions. We then copy the rest of the fragments from Parent2 into the offspring in the relative order presented in Parent2. If the fragment ID is already present in the

offspring, then we skip that fragment. The method preserves the feasibility of every string in the population.

Edge recombination preserves the adjacencies that are common to both parents. This operator is appropriate because a good fragment ordering consists of fragments that are related to each other by a similarity metric and should therefore be adjacent to one another. Parsons [15] defines edge recombination operator as follows:

1. Calculate the adjacencies.
2. Select the first position from one of the parents, call it s.
3. Select s' in the following order until no fragments remain:
   (a) s' adjacent to s is selected if it is shared by both parents.
   (b) s' that has more remaining adjacencies is selected.
   (c) s' is randomly selected if it has an equal number of remaining adjacencies.

Mutation Operator

This operator is used for the modification of single individuals. The reason we need a mutation operator is for the purpose of maintaining diversity in the population. Mutation is implemented by running through the whole population and for each individual, deciding whether to select it for mutation or not, based on a parameter called *mutation rate* (P_m). For our experimental runs, we use the swap mutation operator. This operator randomly selects two positions from a permutation and then swaps the two fragment positions. Since this operator does not introduce any duplicate number in the permutation, the solution it produces is always feasible. Swap mutation operator is suitable for permutation problems like ordering fragments.

Selection operator

The purpose of the selection is to weed out the bad solutions. It requires a population as a parameter, processes the population using the fitness function, and returns a new population. The level of the selection pressure is very important. If the pressure is too low, convergence becomes very slow. If the pressure is too high, convergence will be premature to a local optimum.

In this work, we use ranking selection mechanism [19], in which the GA first sorts the individuals based on the fitness and then selects the individuals with the best fitness score until the specified population size is reached. Note that the population size will grow whenever a new offspring is produced by crossover or mutation. The use of ranking selection is preferred over other selections such as fitness proportional selection [6].

4 DNA Fragment Assembly Problem using the Parallel GA

This section introduces the parallel model that we use in the experiments discussed in the next section. The first part of this section describes the parallel model of GA, while the second part presents the software used to implement that model.

4.1 Parallel Genetic Algorithm

A parallel GA (PGA) is an algorithm having multiple component GAs, regardless of their population structure. A component GA is usually a traditional GA with a single population. Its algorithm is augmented with an additional phase of *communication* code so as to be able to convey its result and receive results from the other components [2].

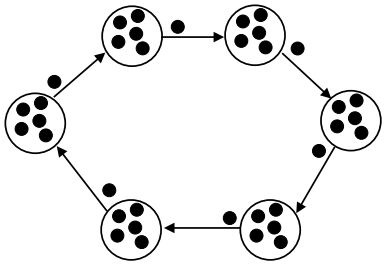

Figure 2: Graphical representation of the parallel dGA

Different parallel algorithms differ in the characteristics of their elementary heuristics and in the communication details. In this work, we have chosen a kind of decentralized distributed search because of its popularity and because it can be easily implemented in clusters of machines. In this parallel implementation separate subpopulations evolve independently in a ring with sparse exchanges of a given number of individuals with a certain given frequency (see Figure 2). The selection of the emigrants is through binary tournament [6] in the genetic algorithms, and the arriving immigrants replace the worst ones in the population only if the new ones is better than this current worst individuals.

Before moving on to Section 5 in which we analyze the effects of several configurations of migration rates and frequencies, we introduce MALLBA which we used in this work and where all our programs can be found.

4.2 The MALLBA Project

The MALLBA research project [1] is aimed at developing a library of algorithms for optimization that can deal with parallelism in a user-friendly and, at the same time, efficient manner. Its three target environments are sequential, LAN and WAN computer platforms. All the algorithms described in the next

section are implemented as *software skeletons* (similar to the concept of software pattern) with a common internal and public interface. This permits fast prototyping and transparent access to parallel platforms.

MALLBA skeletons distinguish between the concrete problem to be solved and the solver technique. Skeletons are generic templates to be instantiated by the user with the features of the problem. All the knowledge related to the solver method (e.g., parallel considerations) and its interactions with the problem are implemented by the skeleton and offered to the user. Skeletons are implemented by a set of *required* and *provided* C++ classes that represent an abstraction of the entities participating in the solver method:

- **Provided Classes**: They implement internal aspects of the skeleton in a problem-independent way. The most important *provided* classes are `Solver` (the algorithm) and `SetUpParams` (setup parameters).
- **Required Classes**: They specify information related to the problem. Each skeleton includes the `Problem` and `Solution` required classes that encapsulate the problem-dependent entities needed by the solver method. Depending on the skeleton other classes may be required.

Therefore, the user of a MALLBA skeleton only needs to implement the particular features related to the problem. This speeds considerably the creation of new algorithms with minimum effort, especially if they are built up as combinations of existing skeletons (*hybrids*).

The infrastructure used in the MALLBA project is made of communication networks and clusters of computers located at the Spanish universities of Málaga, La Laguna and UPC in Barcelona. These nodes are interconnected by a chain of Fast Ethernet and ATM circuits. The MALLBA library is publicly available at `http://neo.lcc.uma.es/mallba/easy-mallba/index.html`.

By using this library, we were able to perform a quick coding of algorithmic prototypes to cope with the inherent difficulties of the DNA fragment assembly problem.

5 Experimental Results

A target sequence with accession number BX842596 (GI 38524243) was used in this work. It was obtained from the NCBI web site (`http://www.ncbi.nlm.nih.gov`). It is the sequence of a Neurospora crassa (common bread mold) BAC, and is 77,292 base pairs long. To test and analyze the performance of our algorithm, we generated two problem instances with GenFrag [5]. The first problem instance, 842596_4, contains 442 fragments with average fragment length of 708 bps and coverage 4. The second problem instance, 842596_7, contains 733 fragments with average fragment length of 703 bps and coverage 7.

We evaluated each assembly result in terms of the number of contigs assembled and the percentage similarity of assembled regions with the target sequence.

Since we obtain fragments from a known target sequence, we can compare our assembled consensus sequence with the target.

We use a sequential GA and several distributed GAs (having 2, 4, and 8 islands) to solve this problem. Since the results of the GA vary depending on the different parameter settings, we begin this section by discussing how the parameters affect the results and the performance of the GA. We then elaborate on how these parameters are used for solving the DNA fragment assembly problem.

5.1 Analysis of the Algorithm

We studied the effects of the fitness function, crossover operator, population size, operator rates and migration configuration for the distributed GA (dGA). In our analysis we perform different runs of the GA in the following manner: change one GA parameter of the basic configuration while keeping the other parameters to the same value. The basic setting uses F1 (Eq. 2) as fitness function and the order-based crossover as recombination operator. The whole population is composed of 512 individuals. In dGA, each island has a population of $512/n$, where n is the number of islands. Migration occurs in a unidirectional ring manner, sending one single randomly chosen individual to the neighbor subpopulation. The target population incorporates this individual only if it is better than its presently worst solution. The migration step is performed every 20 iterations in every island in an asynchronous way. All runs were performed on a Pentium 4 at 2.8 GHz linked by a Fast Ethernet communication network. Our parameter values are summarized in Table 1. We performed 30 independent runs of each experiment.

Independent runs	30
Popsize	512
Fitness function	F1
Crossover	OX (0.7)
Mutation	Swap (0.2)
Cutoff	30
Migration frequency	20
Migration rate	1
Instance	38524243_4.dat

Table 1: Basic Configuration

5.1.1 Function Analysis

We begin our study with the choice of the fitness function because it is one of the most important steps in applying a genetic algorithm to a problem. Especially in this problem, choosing an appropriate fitness function is an open research line, since it is difficult to capture the dynamics of the problem into a mathematical

function. The results of our runs are summarized in Table 2. The table shows the fitness of the best solution obtained (b), the average fitness found (f), average number of evaluations (e), and average time in seconds (t). Recall that F1 is a maximization function while F2 is a minimization one. Our conclusion is that, while F2 takes longer than F1, both functions need the same number of evaluations to find the best solution (differences are not statistically significants in sequential and distributed (n = 2) versions). This comes as no surprise, since F2 has a quadratic complexity while the complexity of F1 is linear. In fact, this amounts to an apparent advantage of F1, since it provides a lower complexity compared to F2 while needing a similar effort to arrive to similar or larger degrees of accuracy. Also, when distributed (n = 4 or n = 8) F1 does allow for a reduction in the effort, while F2 seems not to profit from a larger number of machines.

		F1				**F2**			
		b	f	e	t	b	f	e	t
Sequential		26358	24023	808311	56	55345800	58253990	817989	2.2e+03
LAN	n = 2	98133	86490	711168	25	58897100	61312523	818892	1.1e+03
	n = 4	75824	66854	730777	14	66187200	68696853	818602	5.4e+02
	n = 8	66021	56776	537627	6.3	77817700	79273330	817638	2.7e+02

Table 2: Results with F1 and F2 fitness functions

The number of contigs is used as the criterion to judge the quality of the results. As it can be seen in Table 3, F1 performs better than F2 since it produces fewer contigs.

		F1 Contig	**F2 Contig**
Sequential		6	8
LAN	n = 2	6	7
	n = 4	6	6
	n = 8	6	7

Table 3: Function Analysis 3: Best Contigs

5.1.2 Crossover Operator Analysis

In this subsection we analyze the effectiveness of two recombination operators: the order-based crossover and the edge recombination. Table 4 summarizes the results obtained using these operators. A close look at the columns devoted to running times reveals that ER is slower than the OX operator. This is due to the fact that ER preserves the adjacency present in the two parents, while OX does not. Despite the theoretical advantage of ER over OX, we noticed that the GA performs equally with the order-based crossover operator as with the edge recombination, since it computes higher fitness scores for two out of two cases for the two operators (recall that F1 is a maximization function). OX operator is much faster at an equivalent accuracy.

		OX				ER			
		b	f	e	t	b	f	e	t
Sequential		26358	24023	808311	56	26276	23011	801435	2.01e3
LAN	n = 2	98133	86490	711168	25	99043	84238	789585	1.1e3
	n = 4	75824	66854	730777	14	73542	65893	724058	5.3e2
	n = 8	66021	56776	537627	6.3	62492	53628	557128	1.9e2

Table 4: Crossover Operator Analysis

5.1.3 Population Size Analysis

In this subsection we study the influence of the population in our algorithms. Table 5 shows the average fitness score and the average time for several population sizes. As can be seen in the table, small population sizes lead to fast convergence to low average fitness values. The best average fitness in our experiments is obtained with a population of 512 individuals. For population sizes larger than 512 the execution time is increased while the average fitness does not improve. This observation leads us to believe that a population size of 512 might be optimal.

Popsize	Seq.		LAN					
	$n = 1$		$n = 2$		$n = 4$		$n = 8$	
	f	t	f	t	f	t	f	t
128	8773	1.7	53876	11	16634	3	23118	3.5
256	7424	0.01	76447	29	44846	7	21305	1.9
512	**24023**	**56**	**86490**	**25**	**66854**	**14**	**56776**	**6.3**
1024	21012	60	76263	30	60530	13	47026	7.1
2048	23732	67	54298	32	49049	14	32494	3.3

Table 5: Population Size Analysis

5.1.4 Operator Rate Analysis

We now proceed to analyze the effects of the operator rates in the GA behavior. Table 6 summarizes the results using different combinations of crossover rates (P_c) and mutation rates (P_m). Our findings show that the fitness values tend to increase as the crossover and mutation rate increase. The mutation operator is very important in this problem, since when the algorithm does not apply mutation ($P_m = 0.0$), the population converges too quickly and the algorithm yields very bad solutions.

5.1.5 Migration Policy Analysis

We finish this analysis by examining the effects of the migration policy. More precisely, we study the influence of the migration rate ($rate$) and the migration frequency ($freq$). The migration rate indicates the number of individuals that are migrated, while the migration frequency represents the number of iterations between two consecutive migrations. These two values are crucial for the coupling between the islands in the dGA. Table 7 summarizes the results using different combinations of these parameters. Upon examining the average fitness

P_c-P_m	Seq.			LAN								
	$n=1$			$n=2$			$n=4$			$n=8$		
	b	f	t	b	f	t	b	f	t	b	f	t
0.3-0.0	8842	7817	0	12539	9148	0.2	11500	8775	0.1	10284	7932	0.01
0.3-0.1	15624	12223	33	91989	61549	16	51109	43822	8	34645	29582	3.6
0.3-0.2	22583	17567	33	90691	70342	15	70608	59581	8.1	50336	41728	3.6
0.3-0.3	27466	20476	33	96341	77048	15	78339	66137	7.8	59242	51174	3.6
0.5-0.0	8908	7620	0	28485	12981	0.5	13629	10236	0.2	12788	8522	0.03
0.5-0.1	18103	12600	45	83121	61355	20	54930	47894	11	40523	31871	4.9
0.5-0.2	22706	19038	44	95352	77583	20	73333	62963	11	53326	45378	5
0.5-0.3	28489	23180	45	101172	84300	22	80102	70013	11	59946	53567	5
0.7-0.0	9157	7459	0	28221	12540	0.6	14702	11099	0.2	16089	8935	0.05
0.7-0.1	17140	14065	56	86284	67225	27	59714	50899	14	39862	33719	6.3
0.7-0.2	26358	24023	56	98133	86490	25	75824	66854	14	66021	56776	6.3
0.7-0.3	28359	25026	56	104641	84065	28	84732	71897	14	63482	53212	6.1
1.0-0.0	8692	7505	0	21664	14561	1.1	26294	12888	0.4	16732	9897	0.11
1.0-0.1	19485	16242	74	90815	71252	35	67067	55713	19	42650	33783	7.9
1.0-0.2	27564	22881	74	103231	81300	35	81417	71963	20	56871	50154	8.4
1.0-0.3	**33071**	**27500**	**74**	**107148**	**88653**	**36**	**88389**	**74048**	**18**	**66588**	**58556**	**8.5**

Table 6: Operator Rate Analysis (P_c-P_m)

column (f), we observe that a lower value of migration rate ($rate = 1$) is better than a higher value. A high coupling among islands (a low value of migration frequency) is not beneficial for this problem. The optimum value of migration frequency is 20, since if we still increase this value (resulting in a looser coupling among islands) the average fitness decreases.

$freq$-$rate$	LAN					
	$n=2$		$n=4$		$n=8$	
	b	f	b	f	b	f
5-1	99904	76447	75317	62908	52127	35281
5-10	61910	37738	68703	55071	56987	52128
5-20	92927	72445	72029	66368	59473	54312
20-1	**98133**	**86490**	**75824**	**66854**	**66021**	**56776**
20-10	82619	45375	70497	57898	53941	48968
20-20	89211	74236	72170	65916	59324	53352
50-1	95670	70728	77024	65257	64612	55786
50-10	92678	41465	66046	51321	59013	51842
50-20	95374	76627	72540	62371	59923	52650

Table 7: Migration Policy Analysis ($freq$-$rate$)

5.2 Analysis of the Problem

In this section we report the results aimed at solving the problem as accurately and efficiently as possible.

From the previous analysis, we conclude that the best settings for our problem instances of the fragment assembly problem is a population size of 512 individuals, with F1 as fitness function, OR as crossover operator (with probability 1.0), and with a swap mutation operator (with probability 0.3). The migration in dGAs occurs in a unidirectional ring manner, sending one single randomly chosen individual to the neighbor sub-population. The target population incorporates this individual only if it is better than its presently worst solution. The migration step is performed every 20 iterations in every island in an asynchronous way. A summary of the conditions for our experimentation is found in Table 8.

Independent runs	30
Popsize	512
Fitness function	F1
Crossover	OR (1.0)
Mutation	Swap (0.3)
Cutoff	30
Migration frequency	20
Migration rate	1

Table 8: Parameters when heading and optimum solution of the problem

Table 9 shows all the results and performance with all data instances and algorithms described in this chapter. We discuss some of the results found in the table. First, for both instances, it is clear that the distributed version outperforms the serial version. The distributed algorithm yields better fitness values and is faster than the sequential GA. Let us now go in deeper details on these claims.

For the first problem instance, the parallel GAs sampled less points in the search space than the serial one, while for the second instance the panmictic algorithm is mostly similar in the required effort with respect to the parallel ones.

Increasing of the number of islands (and CPUs) results in a reduction in search time, but it does not lead to a better fitness value. For the second problem instance, the average fitness was improved by a larger number of islands. However, for the first problem instance, we observed a reduction in the fitness value as we increased the number of CPUs. This counterintuitive result clearly states that each instance has a different number of optimum number of islands from the point of view of the accuracy.

The best tradeoff is for two islands (n = 2) for the two instances, since this value yields a high fitness at an affordable cost and time.

		38524243_4				**38524243_7**			
		b	f	e	t	b	f	e	t
Sequential		33071	27500	810274	74	78624	67223	502167	120
LAN	$n = 2$	107148	88653	733909	36	156969	116605	611694	85
	$n = 4$	88389	74048	726830	18	158021	120234	577873	48
	$n = 8$	66588	58556	539296	8.5	159654	119735	581979	27

Table 9: Results of Both Problem Instances

Table 10 gives the speed-up results. As it can be seen in the table, we always obtain an almost linear speedup for the first problem instance. For the second instance we also have a good speedup with a low number of islands (two and four islands); eight islands make the efficiency decrease to a moderate speedup (6.42).

Finally, Table 11 shows the global number of contigs computed in every case. This value is used as a high-level criterion to judge the whole quality

		38524243_4			38524243_7		
		n CPUs	1 CPU	Speedup	n CPUs	1 CPU	Speedup
LAN	$n = 2$	36.21	72.07	1.99	85.37	160.15	1.87
	$n = 4$	18.32	72.13	3.93	47.78	168.20	3.52
	$n = 8$	8.52	64.41	7.56	26.81	172.13	6.42

Table 10: Speed-up

of the results since, as we said before, it is difficult to capture the dynamics of the problem into a mathematical function. These values are computed by applying a final step of refinement with a greedy heuristic regularly used in this application [11]. We have found that in some (extreme) cases it is possible that a solution with a better fitness than other one generates a larger number of contigs (worse solution). This is the reason for still needing research to get a more accurate mapping from fitness to contig number. The values of this table confirm again that all the parallel versions outperform the serial versions, thus advising the utilization of parallel GAs for this application in the future.

		38524243_4	38524243_7
Sequential		5	4
LAN	n = 2	3	2
	n = 4	4	1
	n = 8	4	2

Table 11: Final Best Contigs

6 Conclusions

The DNA fragment assembly is a very complex problem in computational biology. Since the problem is NP-hard, the optimal solution is impossible to find for real cases, except for very small problem instances. Hence, computational techniques of affordable complexity such as heuristics are needed for this problem.

The sequential Genetic Algorithm we used here solves the DNA fragment assembly problem by applying a set of genetic operators and parameter settings, but does take a large amount of time for problem instances that are over 15k base pairs. Our distributed version has taken care of this shortcoming. Our test data are over 77K base pairs long. We are encouraged by the results obtained by our parallel algorithms not only because of their low waiting times, but also because of their high accuracy in computing solutions of even just 1 contig. This is noticeable since it is far from triviality to compute optimal solutions for real-world instances of this problem.

We plan to analyze other kinds of distributed algorithms created as extensions of the canonical GA skeleton used in this chapter. To curb the problem of premature convergence for example, we propose a restart technique in the islands. Another interesting point of research would be to incorporate different algorithms in the islands, such as greedy or simulated annealing, and to study

the effects this could have on the observed performance.

Acknowledgments

The first two authors are partially supported by the Ministry of Science and Technology and FEDER under contract TIC2002-04498-C05-02 (the TRACER project).

References

[1] E. Alba and the MALLBA Group. MALLBA: A library of skeletons for combinatorial optimisation. In R. Feldmann B. Monien, editor, *Proceedings of the Euro-Par*, volume 2400 of *Lecture Notes in Computer Science*, pages 927–932, Paderborn (GE), 2002. Springer-Verlag.

[2] E. Alba and M. Tomassini. Parallelism and Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, October 2002.

[3] C. F. Allex. *Computational Methods for Fast and Accurate DNA Fragment Assembly*. UW technical report CS-TR-99-1406, Department of Computer Sciences, University of Wisconsin-Madison, 1999.

[4] T. Chen and S. S. Skiena. Trie-based data structures for sequence assembly. In *The Eighth Symposium on Combinatorial Pattern Matching*, pages 206–223, 1998.

[5] M. L. Engle and C. Burks. Artificially generated data sets for testing DNA fragment assembly algorithms. *Genomics*, 16, 1993.

[6] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[7] P. Green. Phrap. http://www.mbt.washington.edu/phrap.docs/phrap.html.

[8] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan, 1975.

[9] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[10] S. Kim. *A structured Pattern Matching Approach to Shotgun Sequence Assembly*. PhD thesis, Computer Science Department, The University of Iowa, Iowa City, 1997.

[11] L. Li and S. Khuri. A comparison of dna fragment assembly algorithms. In *International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, pages 329–335, 2004.

[12] E. W. Myers. Towards simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2(2):275–290, 2000.

[13] C. Notredame and D. G. Higgins. SAGA: sequence alignment by genetic algorithm. *Nucleic Acids Research*, 24:1515–1524, 1996.

[14] R. Parsons, S. Forrest, and C. Burks. Genetic algorithms, operators, and DNA fragment assembly. *Machine Learning*, 21:11–33, 1995.

[15] R. Parsons and M. E. Johnson. A case study in experimental design applied to genetic algorithms with applications to DNA sequence assembly. *American Journal of Mathematical and Management Sciences*, 17:369–396, 1995.

[16] P. A. Pevzner. *Computational molecular biology: An algorithmic approach.* The MIT Press, London, England, 2000.

[17] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*, chapter 4 - Fragment Assembly of DNA, pages 105–139. University of Campinas, Brazil, 1997.

[18] G. G. Sutton, O. White, M. D. Adams, and A. R. Kerlavage. TIGR Assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science & Technology*, pages 9–19, 1995.

[19] D. Whitely. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann, 1989.