Computer Science and Information Systems 15(2):273–293 https://doi.org/10.2298/CSIS170510009V

Solving the DNA Fragment Assembly Problem with a Parallel Discrete Firefly Algorithm implemented on GPU

Pablo Javier Vidal[1] and Ana Carolina Olivera[1]

Centro de Investigaciones y Transferencia Golfo San Jorge
CONICET - Universidad Nacional de la Patagonia Austral
Ruta N° 3, 9011, Caleta Olivia, Argentina.
{pjvidal,acolivera}@conicet.gov.ar

Abstract. The Deoxyribonucleic Acid Fragment Assembly Problem (DNA-FAP) consists in reconstructing a DNA chain from a set of fragments taken randomly. This problem represents an important step in the genome project. Several authors are proposed different approaches to solve the DNA-FAP. In particular, nature-inspired algorithms have been used for its resolution. Even they were obtaining good results; its computational time associated is high. The bio-inspired algorithms are iterative search processes that can explore and exploit efficiently the solution space. Firefly Algorithm is one of the recent evolutionary computing models which is inspired by the flashing light behaviour of fireflies. Recently, the Graphics Processing Units (GPUs) technology are emerge as a novel environment for a parallel implementation and execution of bio-inspired algorithms. Therefore, the use of GPU-based parallel computing it is possible as a complementary tool to speed-up the search. In this work, we design and implement a Discrete Firefly Algorithm (DFA) on a GPU architecture in order to speed-up the search process for solving the DNA Fragment Assembly Problem. Through several experiments, the efficiency of the algorithm and the quality of the results are demonstrated with the potential to applied for longer sequences or sequences of unknown length as well.

Keywords: DNA Fragment Assembly Problem, Graphic Processing Units, Parallelism, Firefly Algorithm.

1. Introduction

The Deoxyribonucleic Acid Fragment Assembly Problem (DNA-FAP) consists of: obtaining the correct sequence of the DNA by finding the permutation of fragments that best represents the original DNA chain, given a set of large number (hundreds or even thousands) of DNA fragments with errors. The DNA-FAP is the primary goal in any genome project and the remaining phases strongly depend on the accuracy of the results at this stage. The DNA-FAP is therefore a combinatorial optimization problem that is NP-Hard [29]. Over the past decade, a lot of tools have been invented to automate DNA sequencing. Among these tools, PHRAP [10], TIGR assembler [34], STROLL [3], CAP3 [11], Celera assembler [22] and EULER [29] may be cited. Each one automates fragment assembly using a variety of algorithms, the most widely used being those based on greedy techniques.

The problem of DNA fragment assembly has been tackled with different meta-heuristics in the literature [17,20]. Since they are robust search methods requiring little information for searching effectively in large or poorly understood search spaces they have been proven to outperform the greedy techniques in small instances of the problem. However, the quest for new, more accurate and faster techniques still continues. One of the main problems is scaling up these methods to the size of real organisms.

The Firefly Algorithms (FAs), which were developed by Yang [36,37], are recent bio-inspired algorithms that have achieved outstanding results in various domains [6,18,40]. The FA is a population-based approach based on the flashing patterns and behaviour of fireflies [37]. FAs have some significant advantages over other metaheuristics, such as Genetic Algorithms(GAs) and Particle Swarm Optimizers (PSOs) [9]. A couple of its distinctive advantages are: the automatic subgrouping and its ability to deal with multi-modal problems [37]. Fireflies can randomly subdivide into sub-groups and each group can potentially swarm around a local optimum. All optimal solutions (obviously including the global optimum) can be obtained simultaneously if the number of fireflies is much higher than the number of subgroups [36,37]. In a few years, a lot of research around FA has been done with excellent results in many different fields, such as Power Energy Systems [2,6], Mobile Networks [1] and Permutations Combinatorial Problems [18,32,38]. If we consider large instances of the problem, the evaluation of a DNA-FAP solution can easily require more than several tens of seconds. It may be necessary to perform several hundred of thousands of evaluations in each iteration process since metaheuristics use a population of solutions. So, the computational time can be in the order of hundreds of days. In this context, the Graphics Processing Units (GPUs) is recognized as powerful way of achieving high-performance on long running scientific applications [23]. Nevertheless, parallel Firefly Algorithm represents a new developing research area and, to the best of our knowledge, very little research has been urdertaken with FA on GPU [27,12,35]. Based in our previous work [35], we propose a Parallel Discrete Firefly Algorithm running entirely on GPU (GPU-DFA) for solving the DNA fragment assembly problem. The idea in GPU-DFA design is to create a simple but powerful enough algorithm to adapt itself to the GPU environment. Here we present the characteristics for the main processes of the GPU-DFA. Also, we demonstrate that the proposed optimization technique is quite amenable for massive parallelism in order to obtain significant efficiency and substantial gain times. In order to evaluate our approach, sixteen popular benchmarks have been used [19]. We employ these benchmarks for making a comparison between our method and some of the best methods published. The contributions of the paper can be summarized as follows:

- We define the structure of a parallel DFA to be executed mainly in a GPU and determine the time-consuming operations spent by the GPU-DFA.
- A deep analysis of the behaviour of Discrete Firefly Algorithm design for GPU to solve the DNA assembly problem was carry out.
- We incorporate an exploitation operator in our canonical proposal addressed for this problem, a local search strategy, in order to carry out a fine-tuning of solutions.

In the remainder of this paper, we give a description of the Fragment Assembly Problem in Section 2. Thereafter, Section 3 introduces the canonical DFA and the details of our algorithmic proposal. In Section 6 we describe the experimental settings, including a brief explanation about DNA-FAP instances. Then, an analysis of the results is presented in Section 7. Finally, Section 8 provides the conclusions and also highlights future research directions.

2. The DNA Fragment Assembly Problem

DNA-FAP is one of the fundamental problems in computational molecular biology [29]. This problem involves the combination of partial information from known fragments in order to find a consistent and complete DNA chain. Hence, large DNA strands need to be broken into small fragments for sequencing in a process called shotgun sequencing [30], but this process does not keep either the ordering of the fragments or the portion about where the particular fragment came from. This leads to the DNA fragment assembly problem [14] where these short sequences have to then be reassembled in the right order by using the overlapping portions as landmarks. Most fragment assembly algorithms consist of the following steps:

– **Overlap**: Finding the potentially overlapping portions of fragments.
– **Layout**: Finding the order of the fragments based on similarity scores.
– **Consensus**: Deriving the DNA sequence from the layout.

The overlap problem consists of finding the best match between the suffix of one read and the prefix of another one. The common practice is to filter out pairs of fragments that do not share a significantly long common substring.

Constructing the layout is the hardest step in fragment assembly [20]. The difficulty is encountered when deciding whether two fragments really overlap, i.e., that their differences are caused by sequencing errors or they actually come from two different copies of a repeat. Repeat fragments represent the major challenge for the whole genome shotgun sequencing and make the layout problem very difficult.

In order to evaluate a solution for DNA-FAP, the following function can be defined: for a possible order of l fragments (a permutation) $i = [0, ..., a, a+1, ..., l]$ the Equation 1 shows the value of the sequence i for DNA-FAP.

$$f(i) = \sum_{a=0}^{l-1} w_{a,a+1} \tag{1}$$

where $w_{a,a+1}$ is the pairwise overlap strength of fragments a and $a+1$ [26].

The final consensus step of fragment assembly amounts to correcting errors in the sequence of reads. To measure the quality of a consensus, we can look at the coverage distribution. 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, and it denotes the number of fragments, on average, where a given nucleotide in the target DNA is expected to appear. It is computed as the number of bases read from fragments over the target DNA's length [14]. If no sequencing error is detected at the overlap phase, the process simply finds the longest suffix of one string that matches exactly the prefix of another one. However, when sequencing errors exist, the process searches for the best match but a small percentage of errors still remain (1% to 3%). The only information available during the assembly process is the sequence of bases; therefore, the ordering of the fragments must rely primarily on fragment similarity and on how much they overlap.

Once the fragments have been ordered (layout), the final consensus is generated. This means that a multiple alignment is computed to obtain a consensus sequence that will be used as the final genomic sequence. The quality of a consensus can be measured by its coverage distribution.

The coverage is then computed as the number of bases read from fragments over the length of the target DNA:

$$Coverage = \frac{\sum_{i=1}^{l} \text{length of the fragment } i}{\text{target sequence length}}, \tag{2}$$

where l stands for the number of fragments. Thus, the higher the coverage and the smaller the number of gaps, the better the results. A partial coverage is achieved when it is not possible assemble a given set of fragments into a single contig. More precisely, a **contig** in biology is defined as a layout consisting of contiguous overlapping fragments. Overlapping between adjacent fragments must be greater than or equal to a predefined threshold (i.e., cut-off parameter).

The assembly of DNA fragments into a consensus sequence corresponding to the parent sequence constitutes the DNA-FAP, which is NP-hard [29].

3. Firefly Algorithm

The Firefly Algorithm (FA) is a recent bio-inspired metaheuristic developed by Yang [36]. It is inspired by mimicking the flashing and attraction behaviour of fireflies. In the scheme of Yang the fireflies have the following characteristics:

1. All fireflies are unisexual, so that one firefly is attracted to other fireflies regardless of their sex.
2. Attractiveness is proportional to their brightness. Hence, for any two flashing fireflies, the less bright one will move towards the brighter one (see Fig. 1(a)). The attractiveness decrease as their distance increases. So, the attractiveness has the same value of brightness when the distance between two fireflies is 0. If no one is brighter than a particular firefly, it moves randomly as shown in Fig. 1(b).
3. The brightness of a firefly is affected or determined by the landscape of the objective function.

A canonical FA works with two basic concepts: the variation of light intensity I (brightness), and the firefly attractiveness β between two fireflies [36]. In the simplest case for maximum optimization problems, the brightness I of a firefly is its fitness. In addition, light intensity decreases with the distance from its source, and light is also absorbed in the media, so we should allow the attractiveness to vary with the degree of absorption. For a given medium with a fixed light absorption coefficient γ, the light intensity I varies with the distance r (see Equation 3). As a firefly's attractiveness β is proportional to the light intensity seen by adjacent fireflies, we can now define the attractiveness β of a firefly by Equation 4.

$$I_j(r_{ij}) = I_0 e^{-\gamma r_{ij}} \tag{3}$$

$$\beta_j(r_{ij}) = \beta_0 e^{-\gamma r_{ij}^2} \tag{4}$$

where I_0 is the light intensity (brightness) and β_0 the original attractiveness of firefly at $r = 0$, respectively. Whilst the intensity is referred to as an absolute measure of emitted light by the firefly, the attractiveness is a relative measure of the light that should be

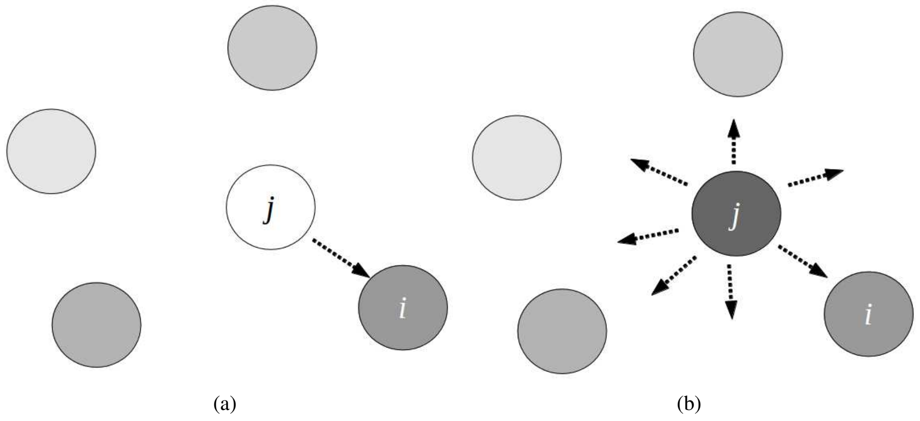

Fig. 1. Firefly movement considering their attractiveness: (a) j moves to i, the brightest firefly close to it; (b) j has more brightness than the most attractive firefly i, so j moves randomly.

seen in the eyes of the beholders and judged by other fireflies [36]. With respect to the light absorption coefficient γ, if $\gamma \rightarrow 0$ the attractiveness of a firefly i matches with its brightness (fitness), i.e., the brightness of a firefly will not decrease when viewed by another one. In the case of $\gamma \rightarrow \infty$, this means that the attractiveness value of a firefly is close to zero when viewed by another firefly in the sense that fireflies fly randomly. So, γ determines the speed of convergence and how the FA behaves and the parameter β controls the attractiveness. Parametric studies suggest that $\beta_0 = 1$ can be used for most applications [39]. The distance between two fireflies i and j which are located at two different locations, can be expressed as a Euclidean distance, as follows:

$$r_{ij} = \|i - j\| = \sqrt{\sum_{k=1}^{k=n} (i_k - j_k)^2} \tag{5}$$

where n denotes the dimensionality of the problem. Taking into account the parameters like r, β and I, FA can define the kind of movement a firefly i can make with respect to a firefly j. A pseudocode explanation of canonical FA can be seen in Algorithm 1. First, in population P all the fireflies are initialized (line 1). Initialize the light intensity of each firefly i with its fitness (line 2). Next, the γ parameter is defined (line 3). Then, while the stop condition is not reached (line 4), for each firefly i, FA tries to find the brightest firefly j near i (line 5 to 13). FA compares I_i and I_j, if $I_j > I_i$ then firefly i will move toward firefly j (line 8). The movement of a firefly i is attracted to another more attractive firefly j is determined by

$$i = i + \beta_0 e^{-\gamma r_{ij}^2} (j - i) + \alpha \epsilon_i \tag{6}$$

where the second term is due to the attraction, while the third term is randomization, with the vector of random variables ϵ_i being drawn from a Gaussian distribution and $\alpha \in [0, 1]$. Then, the Attractiveness is updated (line 10). New solutions are evaluated and the light intensity is updated (line 11). The fireflies are ranked and the current best is found (line 14). When the iterative process ends, FA returns the results obtained (line 16).

Algorithm 1 Canonical Firefly Algorithm.

```
1:  Initialize a population $P$ of fireflies $(i = 1, 2, 3, ..., n)$
2:  Initialize light intensity of each firefly $i$, $I_i = f(i)$
3:  Define light absorption coefficient $\gamma$
4:  while no stopping condition is satisfied do
5:      for $i = 1 : n$ all fireflies do
6:          for $j = 1 : n$, with $i \neq j$ do
7:              if $I_j > I_i$ then
8:                  Move firefly $i$ towards $j$
9:              end if
10:             Attractiveness varies with distance $r$ via $e^{-\gamma r^2}$
11:             Evaluate new solutions and update light intensity
12:         end for
13:     end for
14:     Rank the fireflies and find the current best
15: end while
16: return Inform results
```

As can be seen from Algorithm 1 where it has an inherent $O(n^2 \times t)$ complexity, where t is the number of iterations of the *while*, since every firefly i must evaluate Equation 4 n times, for every other firefly j. This complexity is not easy to reduce.

3.1. Discrete Firefly Algorithm

Discrete FA (DFA) is a variation of canonical FA used for combinatorial problems with success for diverse problems [6,13,25,32]. In the Section 5, the discrete firefly algorithm approach is described in the context of its GPU approximation.

4. Graphic Processing Units

The model for GPU computing uses a CPU and GPU together in a heterogeneous co-processing computing model, the CPU is considered as the host and the GPU is used as the device coprocessor. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU. GPUs may contain several hundred simple processors. The set of processors are usually considered as a Single Instruction Multiple Data (SIMD) computer. A GPU is used by means of a *kernel* that is a function callable from the host and executed on the device by each thread. Compute Unified Device Architecture (CUDA) [4] is an extension of the C programming language and was created by nVidia. CUDA is a C language environment that provides services to programmers and developers ranging from common GPU operations in the CUDA library to traditional C memory management semantics in the CUDA run-time.

Two important concepts with CUDA programming are thread batching and management of the memory model [4]. The host launches a kernel to be executed by a batch of threads on the GPU. The batch of threads can be organized as a grid of thread blocks. A thread block is a batch of threads that can cooperate together by efficiently sharing data through diverse levels of memory and synchronizing their execution to coordinate memory accesses.

5. GPU Discrete Firefly Algorithm

The goal of this section is to present our algorithmic proposal, which has been called GPU-DFA. Our primary concern when designing DFA accelerated by GPU is to establish an efficient model that runs the main processes of DFA entirely on GPU. The objectives are: (1) to minimize the data transfers between the CPU and the GPU, thus avoiding communication bottlenecks; (2) we will also aim to support n-dimensional optimization problems using the firefly model interaction with the chance that our algorithm can support large numbers of fireflies due to optimized data-structures. The CUDA software model is employed so as to exploit maximum parallel execution and high arithmetic intensity of GPUs [24].

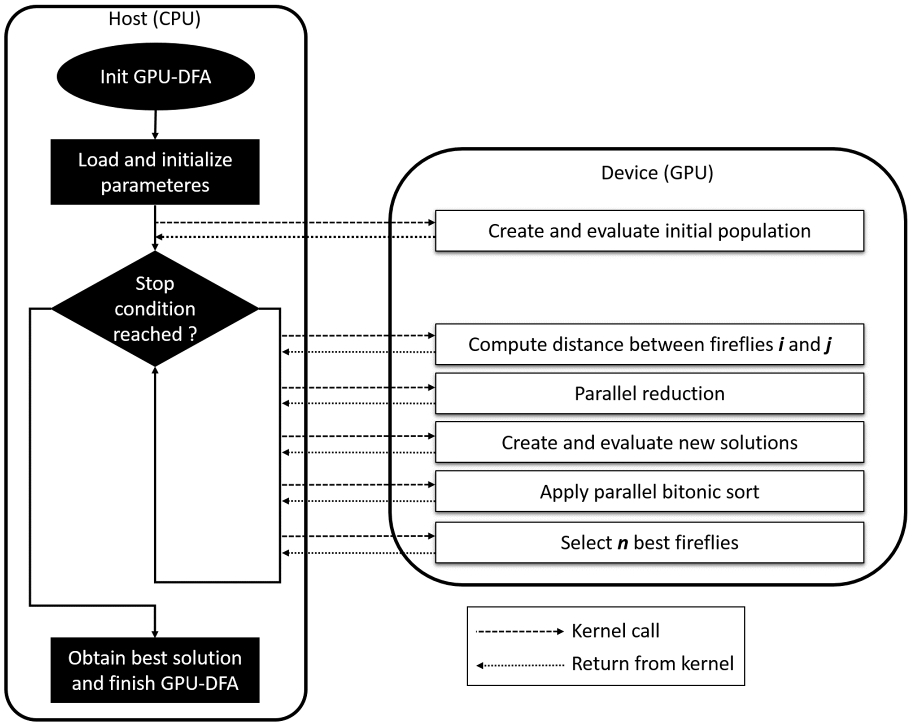
Host (CPU)
Init GPU-DFA
Load and initialize parameteres
Stop condition reached ?
Obtain best solution and finish GPU-DFA
Device (GPU)
Create and evaluate initial population
Compute distance between fireflies *i* and *j*
Parallel reduction
Create and evaluate new solutions
Apply parallel bitonic sort
Select *n* best fireflies
Kernel call
Return from kernel

Fig. 2. GPU Discrete Firefly Algorithm Model.

The flowchart of the GPU-DFA model is presented in Fig. 2. The major resource-consuming part is computed by the GPU and the rest is handled by the CPU. Adapting this kind of bio-inspired algorithm is not a straightforward task because hierarchical memory management on GPU has to be handled. Memory transfers from CPU to GPU are slow and these copying operations have to be minimized. Our proposal starts with the initialization of the FA parameters. All parameters are transferred to the GPU main memory. Then, GPU-DFA creates and evaluates in each thread a firefly solution from P in each thread. Next, we use a group of kernels that execute several tasks until the stop condition has been reached: compute distance between fireflies, parallel reduction, evaluate new solutions, apply parallel sort and select best fireflies. The division in multiple kernels is due to the heterogeneity of the tasks and the complexity thereof.

Algorithm 2 shows the organization of the different processes for the GPU-DFA. The following sections explain each procedure one by one in detail.

Algorithm 2 GPU Firefly Algorithm.

```
1: Define light absorption coefficient $\gamma$
2: Allocate problem inputs on GPU memory
3: Copy problem inputs on GPU memory
4: Allocate solution structure population on GPU memory
5: Initialize a population $P$ of $n$ fireflies
6: for each $firefly$ $(i)$ of $P$ in parallel do
7:    initializeSolution($i$);
8:    evaluateSolution($i$);
9: end for
10: while non stop condition do
11:    temp=$\emptyset$
12:    for each pair $i$ and $j$ from $P$ in parallel do
13:       $A$=computeDifferences($i$, $j$);
14:       $r_{ij}$=computeDistance($i$, $j$,$A$);
15:       $\beta_{ij}$=computeAttractiveness($r_{ij}$,$i$,$j$);
16:    end for
17:    apply reduction function
18:    for all $j \in P$ in parallel do
19:       if $j$ is attracted to other firefly $i$ then
20:          $k = random(2, A)$
21:          temp.add($move_{2-opt}(j, k)$)
22:       else
23:          temp.add($move_{Random}(j)$)
24:       end if
25:    end for
26:    $P$ = Take best $n$ fireflies in temp
27: end while
28: return best firefly in $P$
```

5.1. Parallel initialization

First of all, problem information and additional structures associated with the operation must be copied on GPU. It is important to notice that data inputs are read-only structures and never change during the whole execution of GPU-DFA. Therefore, the copy is only performed once during the whole execution.

Since GPUs require parallel massive computations with predictable memory accesses, it is necessary to allocate a structure for storing all the solutions.

In our GPU-DFA approach, the initialization and evaluation of each solution will be achieved one-by-one. Each firefly i is created randomly (line 7) by permutation of DNA fragments. Next (line 8), the firefly i is evaluated and its fitness (Brightness) is calculated. Hence, GPU-DFA computes them through parallel threads. A set of consecutive threads can access successive memory spaces. In this way we try to follow recommendations and patterns for memory coalescence [15] (lines 6-9).

5.2. Computing distances, Attractiveness and Reduction Operation

For each evolution step, GPU-DFA computes r_{ij}, β, and I in parallel between each pair of fireflies i and j. When evaluating each combination, it is necessary to run a parallel reduction kernel in order to define whether solution j performs a random movement or moves closer to a brighter solution (lines 12-15).

GPU-DFA uses an auxiliary structure located in the shared memory for storing the partial values computed by each thread. The algorithm launches n block of threads to evaluate the values stored in the shared memory (line 16).

In continuous optimization problem, distance between two fireflies is simply calculated using Euclidian distance. For the DNA-FAP the distance between two fireflies, r_{ij} is defined by Equation 7 (line 14). In this equation, A is the number of different edges between fireflies i and j through the number of consecutive differences in the array positions (line 13). The parameter l indicates the size of problem (number of fragments). In Equation 7, $r_{ij} \in [0, 10]$ and it will be used in attractiveness calculation (line 15) [13].

$$r_{ij} = \frac{A}{l} \times 10 \tag{7}$$

In order to clarify A, considerer the Figure 3. It shows the part of fireflies i and j with differences in the order of DNA fragments. The value of A is five, the different edges between them are $(2, 4), (4, 5), (7, 1), (10, 3)$ and $(3, 6)$.

The attractiveness is calculated exactly as Canonical FA. The light intensity I of a firefly i is define as the fitness value of the firefly.

5.3. Parallel Modifications

Once each firefly j has a defined movement, GPU-DFA creates and evaluates $n \times m$ new solutions (by disturbing each one with a specific operator or randomly and saves them in $temp$ (lines 17-24). m is number of new fireflies created considering each firefly j movement. In order to explore the solution space, GPU-DFA uses a 2-opt movement for

order	8	9	10	11	12	13	14	15	16	17
i	…	2	4	5	7	1	10	3	6	…
j	…	2	5	7	4	1	10	6	3	…

Fig. 3. Fragment of fireflies i and j, A is calculated for two fireflies i and j as the number of different edges between them.

DNA-FAP. The movement is applied k times (line 21), where k is a number selected randomly between 2 and parameter A (line 20). Otherwise, random movement is generated by applying a 2-opt operator without any restrictions (line 23).

5.4. Bitonic Sort

The firefly scheme needs to sort the $temp$ according to its fitness by using parallel Bitonic Sort [28] to get the n best fireflies to replace on P. Bitonic sort has primarily been used by previous GPU sorting algorithms even though the classic complexity is of $n(\log n)^2$. This method needs a total number of comparison/exchange operations of $O(n \log n)$. The hidden constant in the asymptotic is smaller than in other parallel sorting methods. It can be implemented in a highly parallel manner on modern architectures, even without any scatter operations, that is, without random access writes, which would involve passing a sequential execution mode within the GPU. Next, select n solutions in parallel to replace the population P (line 26).

5.5. Random Number Generation

The performance of a nature-inspired algorithm largely depends on the quality of its random number generator across the entire evolution. We used a Mersenne Twister random generator approach [31] for this study. At the beginning of the execution, GPU-DFA set a global seed that is passed and it is used to set up one local seed per thread. Finally, this local seed is invoked continuously by each thread for subsequent random number generation. This approach has been successfully tested in other work [35].

5.6. Solution Encoding

In order to make an efficient mapping between a thread and a particular solution, we use a discrete vector representation. This codification uses an alphabet $\Sigma = \{1, \ldots, l\}$ where l is the total number of fragments to be sorted. Then, a solution is a permutation of fragments of DNA. In this representation, each variable takes its value over the finite alphabet Σ.

The memory layouts of the GPU population are carefully designed. The chromosome-based layout [33] simplifies the solution movements in the selection, comparison, perturbation phases, and replacement phases in order to preserve the data locality.

5.7. GPU-DFA exploitation: Local Search

In addition to improve the model of canonical GPU-DFA, we incorporate other proposal with a Local Search defined as 3-opt movement. This process is performed after line 26 in Algorithm 2 for all the fireflies in P. The 3-opt analysis involves deleting 3 connections (or edges) in a DNA sequence, reconnecting the sequence in all other possible ways, and then evaluating each reconnection method to find the optimum one. This process is then repeated for a different set of 3 edges at random.

6. Experimental settings

In this section we present our experimental set-up. First, we show the features of the selected DNA-FAP instances. Next, the methodology and the parameter settings used in the tests are summarised.

6.1. Instances

We carried out several experiments with different instances of DNA-FAP benchmark data sets, which were described in [19].

Table 1. Information of datasets

Instances (Acronym)	Coverage	Mean fragment length	Num. of fragments	Original sequence length	Optimum
GenFrag Instances					
x60189_4 (x_4)	4	395	39	3835	11478
x60189_5 (x_5)	5	386	48		14161
x60189_6 (x_6)	6	343	66		18301
x60189_7 (x_7)	7	387	68		21271
m154216_5 (m_5)	5	398	127	10089	38746
m154216_6 (m_6)	6	350	173		48052
m154216_6 (m_7)	7	383	177		55171
j02459_7 (j_7)	7	405	352	20000	116700
bx842596_4 (b_4)	4	708	442	77292	227920
bx842596_7 (b_7)	7	703	773		445422
DNAgen Instances					
acin1 (a_1)	26	182	307	2170	47618
acin2 (a_2)	3	1002	451	147200	151553
acin3 (a_3)	3	1001	601	200741	167877
acin5 (a_5)	2	1003	751	329958	163906
acin7 (a_7)	2	1003	901	426840	180966
acin9 (a_9)	7	1003	1049	156305	344107

The benchmarks can be divided in two groups, the first one was generated using *GenFrag* [5] and the *DNAGen* program [19] was used for the second one. Table 1 summarizes the instances information with their names, coverage, sizes (*mean fragment length* and *number of fragments* per instance), original sequence length and the *optimum*, respectively. The instances are ordered by group and number of fragments. A more detailed description of these benchmarks can be found in [19]. In order to test the GPU-DFA we use $n = 32$ (number of fireflies) and $m = 16$ (number of new solutions generated) as parameter settings. These values were obtained from a previous study presented in [35].

6.2. Experimentation

In order to analyse both the behaviour and performance of our approach, we need to clarify some parameter definitions and mechanisms.

In order to make a meaningful comparison between both the DFA implementations -CPU and GPU-, we have chosen a stop condition so that we can guarantee a similar exploration of the search space for all the instance problem sizes. The stop condition for both algorithms is defined in one million of evaluations.

We perform 30 independent runs to evaluate the instances. We have marked a result in dark grey when it is the best and in light grey when it is the second best result.

The experiments were performed using the host with a CPU AMD FX(tm)-8320 Eight-Core Processor, with a total physical memory of 16GB. The operating system is Ubuntu Precise 12.04. In the case of the GPU, we have an NVIDIA GeForce GTX 780Ti with 3GB of DRAM on the device and we used the CUDA version 6.0.

7. Results and Analysis

This section presents the experimental results obtained with the GPU-DFA approach and GPU-DFA with Local Search (GPU-DFA+LS). We measured the quality of the solution by considering the fitness value achieved as well as the number of contigs obtained solving different DNA-FAP instances. Next, the time performance of GPU-DFA and GPU-DFA+LS was analysed and compared with respect to the CPU-DFA algorithm. Finally, an analysis for each GPU-DFA kernel time execution was provided.

7.1. Numerical Analysis

Table 2 shows the results of GPU-DFA approaches for all the DNA-FAP instances. The first column presents the acronym of each instance. Column two shows the best fitness value obtained and column three the hit-rate (percentage of successful runs that obtain the optimum). Columns four and five indicate the average fitness for 30 runs with its standard deviation. The same results are presented from column six to nine for the GPU-DFA+LS.

From Table 2 we observe that the two approaches obtain the optimal value, or near to it, in most of the instances considered. GPU-DFA finds the optimal value at least once in the small instance sizes. The GPU-DFA achieves the optimum in 6 of 16 instances. For the DNAgen instances, our approach does not reach the optimum with the biggest instances. The GPU-DFA+LS obtain the optimum in 8 of 16 instances. Between both approaches, the one that uses LS obtains results much closer to the optimum value (highest) in the rest of instances. The adddition of a LS improve all the results of the canonical GPU-DFA.

Table 2. GPU-DFA results for DNA-FAP instances.

Instance	*GPU-DFA*				*GPU-DFA+LS*			
	Best	Hit-rate	Avg.	Std.	Best	Hit-rate	Avg.	Std.
GenFrag Instances								
x_4	11478	7	11477.67	±0.72	11478	30	114778.00	±0.00
x_5	14027	24	13783.33	±1.09	14161	28	14100.36	±0.74
x_6	18301	25	18297.40	±1.27	18301	28	18299.27	±0.58
x_7	21271	28	21267.83	±0.41	21271	30	21271.00	±0.00
m_5	38592	18	38585.80	±4.37	38746	26	38715.85	±2.61
m_6	48048	13	47969.29	±11.79	48052	18	47969.78	±3.05
m_7	55067	0	54748.50	±58.50	55072	0	54921.09	±13.37
j_7	114358	0	114343.60	±59.45	116700	2	116119.93	±19.36
b_4	225858	0	225173.60	±678.54	227233	0	225173.46	±408.88
b_7	441992	0	433958.00	±753.02	444162	0	443719.33	±102.94
DNAgen Instances								
a_1	47618	1	45976.60	±28.62	47618	8	47576.07	±15.66
a_2	144634	0	144513.00	±33.93	144705	0	144535.79	±69.13
a_3	156776	0	155751.30	±147.10	162961	0	161453.53	±515.69
a_5	147880	0	145304.50	±435.86	160227	0	159304.47	±286.26
a_7	157032	0	156439.90	±317.15	168025	0	167582.23	±152.67
a_9	329015	0	328201.70	±536.77	335488	0	334868.20	±215.29

The best solutions obtained by GPU-DFA are competitive and close to the optimum. The results indicate that GPU-DFA is capable of exploring the search space in an effective way showing promising behaviour.

Table 3 summarizes the results obtained by the GPU-DFA+LS and others DNA-FAP assembler algorithms [7,16,17,21]. Values in **bold** with gray background indicate that an algorithm reaches the optimal solution for the instance and values in *italic* with gray background show values with the second best solution found.

From Table 3, we observe that the algorithms with a more intelligent behaviour benefits the search process, obtaining in general competitive results for all the instances. For small and medium benchmarks, the efficiency of all the algorithms is roughly comparable. Our approach obtain the optimum or stay very near being secondly.

Table 3 indicates that for larger instances, the behaviour is similar, obtaining values very near to the optimum as is the case of DNAgen instances where only the PPSO+DE overcome GPU-DFA+LS results. In the case of GenFrag instances, GPU-DFA+LS was superior to the other algorithms. In general, the second best solution is obtained by the GPU-DFA+LS in large instances. The simulation results demonstrate that the proposed GPU-DFA+LS optimizes the overlap score for the 16 benchmark instances tested.

7.2. Comparison of the contigs

This section presents a comparison of our approach with other well-known assembler algorithms. For this analysis we focus on the significance of the contigs. The contig calculation ensures that the best solution obtained represents a continuously assembled sequence.

Table 4 shows the contig solution obtained with the two GPU-DFA proposals compared to other algorithms. These ones are used to solve the DNA-FAP:Artificial bee

Table 3. Results for the 16 benchmark datasets along with GPU-DFA results for the addit.

Instance	*GPU-DFA+LS*	*PPSO+DE*	*QEGA*	*SA*	*PALS*	*SAX*	*POEMS*
			GenFrag Instances				
x_4	**11478**	**11478**	**11476**	**11478**	**11478**	**11478**	**11478**
x_5	**14161**	13642	*14027*	*14027*	14021	*14027*	–
x_6	**18301**	**18301**	*18266*	**18301**	**18301**	**18301**	–
x_7	**21271**	20921	21208	**21271**	21210	*21268*	21261
m_5	**38746**	38686	38578	38583	38526	**38726**	38610
m_6	**48048**	*47669*	47882	**48048**	**48048**	**48048**	–
m_7	*55072*	54891	55020	55048	55067	*55072*	**55092**
j_7	**116700**	114381	116222	116257	115320	115301	*116542*
b_4	*227233*	224797	**227252**	226538	225783	223029	*227233*
b_7	*444162*	429338	443600	436739	438215	417680	**444634**
			DNAgen Instances				
a_1	**47618**	*47264*	47115	46955	46876	46865	–
a_2	*144705*	**147429**	144133	*144705*	144634	144567	–
a_3	*162961*	**163965**	156138	156630	156776	155789	–
a_5	*160227*	**161511**	144541	146607	146594	145880	–
a_7	*168025*	**180052**	155322	157984	158004	157032	–
a_9	*335488*	**335522**	322768	324559	325930	314354	–

colony (ABC) [8], Queen Bee Evolution Based on Genetic Algorithm (QEGA) [16], Simulated Annealing (SA) [7], Problem Aware Local Search (PALS), and Genetic Algorithm [7].

For GenFrag instances, GPU-DFA obtains an optimal layout of the DNA sequence. In the case of the largest DNAgen instances, GPU-DFA cannot find the best layout. However, PALS, SA and GPU-DFA return similar fitness values. So, we can infer that canonical GPU-DFA needs a better exploitation operation to find the best contigs. In fact, the results obtained by the GPU-DFA+LS prove this point. The results of our proposals are competitive with respect to other approaches presented in the literature. Nevertheless, in the case of GPU-DFA, for instances m_7, a_1, a_7 and a_9 as the contig value did not improve the results obtained by PALS and SA. The GPU-DFA+LS enhances the contig value obtained in m_7, a_1 and a_9.

From Table 4 we can conclude that the use of an intelligent search method for search space exploitation obtains better results (a lower number of contigs) than a random seeding or other bio-inspired techniques.

As a final remark, the numerical results of GPU-DFA indicate that the algorithm is able to generate accurate solutions and it explores the search space effectively, so as to identify the region where the optimal solution is located.

Table 4. Best final number of contig for our assembler (using the best configuration) and for other specialized systems.

Instance	GPU-DFA	GPU-DFA+LS	*ABC*	*QEGA*	*SA*	*PALS*	*GA*
GenFrag Instances							
x_4	1	1	1	1	1	1	1
x_5	1	1	1	1	1	1	1
x_6	1	1	1	1	1	1	1
x_7	1	1	1	1	1	1	1
m_5	1	1	3	1	1	1	1
m_6	1	1	2	1	1	1	2
m_7	2	1	2	1	1	1	2
j_7	1	1	3	1	1	1	1
b_4	1	1	12	8	1	1	6
b_7	1	1	12	3	1	1	3
DNAgen Instances							
a_1	2	1	8	4	1	1	5
a_2	1	1	237	233	1	1	236
a_3	1	1	362	358	1	1	358
a_5	1	1	556	552	1	1	522
a_7	722	722	726	722	1	1	722
a_9	552	1	552	552	1	1	552

7.3. Execution Time Analysis

Table 5 provides the average time consumed in seconds for the CPU and GPU implementation of DFA. The table shows the acronym of each instance in the first column. Column two and three indicate the average time for the CPU-DFA and GPU-DFA models. Finally, column four provides the gain time computed.

Concerning the amount of gain time obtained, we have computed this metric by dividing the average time of the CPU-DFA over the average time for the GPU-DFA. As an initial observation, we can say that the gain time is above the value 1.00 indicating that the CPU always spent more execution time than the GPU.

Table 5 illustrates that while the number of fragments increased, the GPU-DFA times did not rise significantly, in contrast with the times of the CPU-DFA version.

Table 5 indicates that the GPU-DFA algorithm obtained lower times in all the instances. The gain time for the GPU-DFA ranges from $1.73\times$ to $9.88\times$. We can observe that the largest value differences exist in medium/large instances. For the case of the integration of a Local Search to the GPU model, we can see that runtimes are increased but not significantly. The results of execution tiems are better (less) than those obtained by the implementation in CPU. With respect to the gains of times obtained, they do not surpass the canonical model GPU-DFA.

These time differences of both GPU approaches with respect to the CPU model might be due to the intrinsic characteristics of certain operations in the DFA model, which have a high degree of parallelization that can maximize the efficiency of each thread and thus, the simplicity of each kernel is maintained. By considering the time performance, we can say that GPU-DFA has a balanced behaviour. GPU-DFA is able to reach, or be close to, the optimal solution in the shortest execution times.

Table 5. Mean time (in seconds) to find the best fitness obtained for each implementation in all instances.

Instance	*CPU-DFA*	*GPU-DFA*	*Gain Time*	*GPU-DFA+LS*	Gain Time
			GenFrag Instances		
x_4	4.13	2.38	1.74	2.76	1.50
x_5	5.58	2.33	2.39	2.70	2.06
x_6	18.58	2.33	7.97	2.80	6.65
x_7	21.59	3.82	5.65	4.47	4.83
m_5	39.12	7.20	5.43	8.35	4.68
m_6	52.37	5.16	10.15	8.77	5.97
m_7	55.21	6.71	8.23	9.39	5.88
j_7	186.31	34.37	5.42	39.18	4.76
b_4	393.18	139.73	2.81	159.29	2.47
b_7	502.66	140.41	3.58	160.07	3.14
			DNAgen Instances		
a_1	171.98	29.54	5.82	34.86	4.93
a_2	382.93	134.58	2.85	158.80	2.41
a_3	486.36	136.95	3.55	161.60	3.01
a_5	719.58	157.91	4.56	186.33	3.86
a_7	1156.12	138.18	8.37	163.05	7.09
a_9	1418.25	143.57	9.88	169.41	8.37

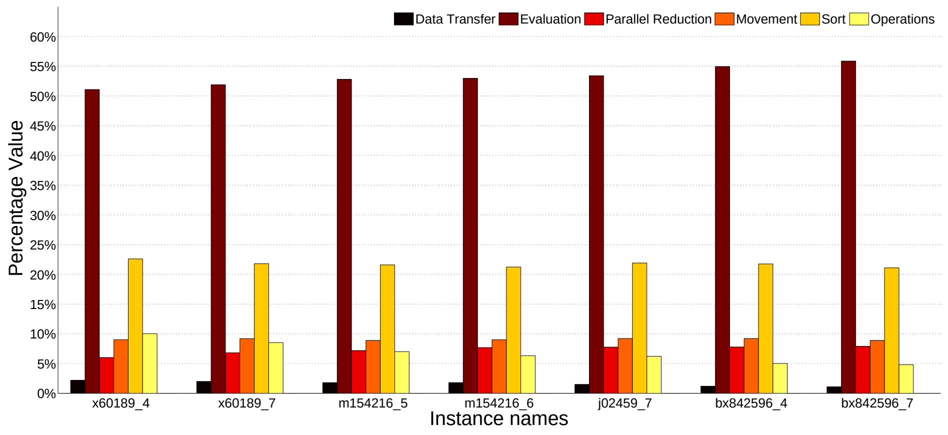

Fig. 4. Average percentage of six time-consuming operations spent by the GPU-DFA model for six DNA-FAP instances.

An important point is the relative execution time spent on GPU-DFA processes with respect to the total execution time of the algorithm. Indeed, these times can help to describe the use of each GPU-DFA process impact over an execution. Fig. 4 and Fig. 5 show the percentage of time spent by each relevant operation in the GPU-DFA and GPU-DFA+LS implementation, respectively. In particular, we focus on: the data transfer be-

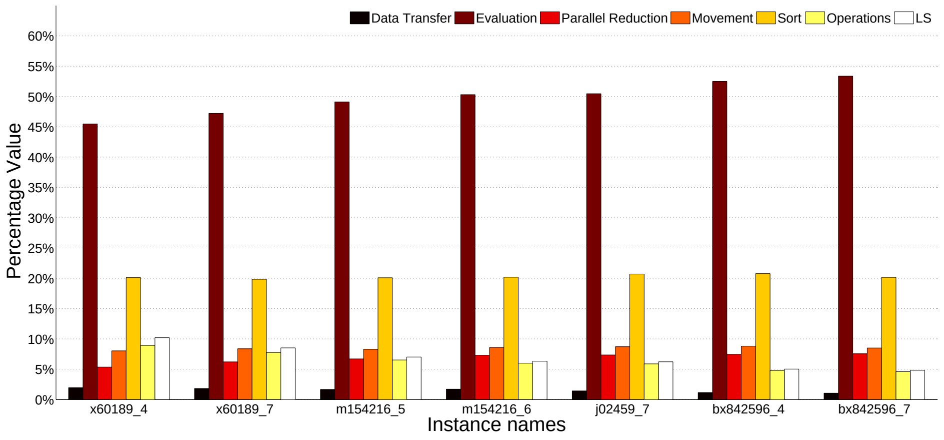

Fig. 5. Average percentage of seven time-consuming operations spent by the GPU-DFA+LS model for six DNA-FAP instances.

tween CPU and GPU, evaluation, reduction, sort, generation of movement, and other minor operations (such as computing distance between two fireflies and replacing solutions between $temp$ and P). Fig. 2 presents the GPU-DFA operations that are analysed. In addition, Fig. 5 shows the time used by the local search procedure in GPU.

A first observation that can be made on Fig. 4 is about the data transfers between the CPU and the GPU. The time associated with the transfers is less than 3% and is kept for all instance sizes. Indeed, from the first instance ($x60189\_4$), the time corresponds to 2.60% and it reaches 1.47% for the last instance ($bx842596\_7$). The time dedicated to the data transfers in the iteration-level parallel model is not significant in comparison with the rest of the processes. Another observation concerns the time spent by the sort and the evaluation of the solutions which represents most of the total running time. The evaluation process time grows accordingly with the instance size (more than 50% time is busy with this process).

Also, in Fig. 4 we can appreciate the time-consuming sort process. The time consumed by the sort process ranged between 23% and 21% of the total running time. In this sense, we can see that as time sorting decreases, the evaluation process grows. These values can be explained by taking into account that, as the number of fireflies increases, the number of iterations decreases. So, the evaluation process needs more time to work with each solution (since the number of elements to be analysed grows). With fewer iterations the evaluation process consumes almost 55% of the running time in the largest instance. Moreover, we observe that the parallel reduction time grows by a smaller proportion when instances become larger (ranging between 8% and 9%).

For the Fig. 5, it can be seen that the execution times obtained are similar to those achieved in Fig. 4. However, it is also notorious how the LS process consumes approximately 10% of the total execution time. To conclude with the analysis of Fig. 4 and Fig. 5, the advantage of a full distribution of processes over different GPU kernels shows that even when wasting time in the data transfer operations and the sort function, the algorithm is still fast and effective. The case of the data transmission performed by the algorithmic model is less time consuming. In regard to the other GPU-DFA processes, we can see that those related with the main FA model have very similar time cost during GPU occupation. This indicates that this model can distribute tasks in different kernels and no single task slows down the system more than others (with the exception of the evaluation process which for all instances demands the longest time).

As final remarks regarding our approach, these results demonstrate that GPU-DFA obtains satisfactory results. We have confidence in the algorithmic performance since we have not yet introduced any specific function or operator related to DNA-FAP in particular.

8. Conclusions and Future Work

In this paper, we have proposed a Discrete Firefly Algorithm implemented on GPU to assemble DNA fragments in order to reconstruct a genome sequence. The algorithm was executed on a GPU platform designed for massive parallel arithmetic computing. The new algorithm is inspired by the successful experience gathered by the FA in different fields. We performed the tests over a group of well-known DNA-FAP benchmarks and compared it with others assemblers. Promising results were obtained considering not only the optimal values, but also the number of contigs generated by GPU-DFA.

From the analysis of all the results we can observe that the GPU-DFA can achieve the optimum or is very close to it but cannot find the best layout for largest DNAgen instances. So, we propose a GPU-DFA enhancement approach (GPU-DFA+LS) in order to exploit the neighbourhood of these solutions. The preliminary results of our experiments are very promising in regard to the effectiveness of the method that we have developed. Indeed, GPU-DFA+LS can be considered a satisfactory assembler, in view of its high accuracy and real time efficiency for DNA assembler sequence problems. The GPU-DFA+LS seems to be a favourable optimization tool in part due to the effect of the attractiveness function and its search model.

In general, both approaches provide a good balance of exploitation and exploration. The implementation on a parallel platform allows the optimal values to reached in a short time, which is interesting, with the idea of applying this algorithm for big database of genomes. Moreover, we have found that the GPU-DFA model provides a robust parallel model that allows instances of different sizes to be solved without great degradation in the quality of the solutions and time execution.

Another conclusion that can be easily observed from the results, is the better scalability of the GPU-DFA approach with respect to the execution time. The gain time of our model in front of the CPU-DFA ranges between $1.50\times$ to $9.88\times$. Furthermore, communication times are quite short compared with the time processes of the GPU-DFA model, so, the numerical performance is not penalized by the communication time. However, this may be a direct effect of the number of steps that the algorithm executes. The GPU-DFA and GPU-DFA+LS obtain similar execution times making the proposals capable to distribute effectively the task in different kernels. It is probe that hybridizing the GPU model could enhance the quality of results without resounding impact on the execution time.

Considering comparison with others assemblers, it is clear that we need to explore the possibility of an intelligent initialization of the fireflies in future work in order to generate quality initial solutions.

Acknowledgments. Authors thank to Agencia Nacional de Promoción Científica y Tecnológica for grant PICT 2014-0430 and grant PIP 2015-2017 from Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET). Authors are thankful to Universidad Nacional de la Patagonia Austral (UNPA). In the same way, the work of Pablo Javier Vidal and Ana Carolina Olivera has been partially funded by the PI-UNPA 29/B209.

References

1. Bojic, I., Podobnik, V., Ljubi, I., Jezic, G., Kusek, M.: A self-optimizing mobile network: Auto-tuning the network with firefly-synchronized agents. INFORM SCIENCES 182(1), pp. 77–92 (2012)
2. Chandrasekaran, K., Simon, S.P.: Network and reliability constrained unit commitment problem using binary real coded firefly algorithm. INT J ELEC POWER 43(1), pp. 921–932 (2012)
3. Chen, T., Skiena, S.S.: A case study in genome-level fragment assembly. Bioinformatics 16 (2000)
4. Corporation, N.: NVIDIA CUDA C Programming Best Practices Guide. Tech. rep. (2009)
5. Engle, M.L., Burks, C.: Artificially generated data sets for testing DNA sequence assembly algorithms. Genomics 16(1), pp. 286–288 (1993)
6. Farhoodnea, M., Mohamed, A., Shareef, H., Zayandehroodi, H.: Optimum placement of active power conditioners by a dynamic discrete firefly algorithm to mitigate the negative power quality effects of renewable energy-based generators. International Journal of Electrical Power & Energy Systems 61, pp. 305–317 (2014)
7. Firoz, J.S., Rahman, M.S., Saha, T.K.: Bee algorithms for solving dna fragment assembly problem with noisy and noiseless data. In: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation. pp. 201–208. ACM, New York (2012)
8. Firoz, J.S., Rahman, M.S., Saha, T.K.: Bee algorithms for solving dna fragment assembly problem with noisy and noiseless data. In: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation. pp. pp. 201–208. ACM, New York (2012)
9. Fister, I., Jr., I.F., Yang, X.S., Brest, J.: A comprehensive review of firefly algorithms. CoRR abs/1312.6609 (2013)
10. Green, P.: Phrap, version 1.090518, http://phrap.org (2009)
11. Huang, X., Madan, A.: CAP3: A DNA sequence assembly program. Genome research 9(9), pp. 868–877 (1999)
12. Husselmann, A., Hawick, K.: Parallel parametric optimisation with firefly algorithms on graphical processing units. In: Hamid (ed.) 2012 World Congress in Computer Science, Computer Engineering, and Applied Computing (2012)
13. Jati, G.K., Manurung, R., Suyanto: Discrete firefly algorithm for traveling salesman problem: A new movement scheme. In: Yang, X.S., Cui, Z., Xiao, R., Gandomi, A.H., Karamanoglu, M. (eds.) Swarm Intelligence and Bio-Inspired Computation, pp. 295–312. Elsevier, Oxford (2013)
14. Jones, N.C., Preface, P.A.P.: An Introduction to Bioinformatics Algorithms. Massachusetts Institute of Technology (2004)
15. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2010)
16. Knuth, D.E.: The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)

17. Kubalik, J., Buryan, P., Wagner, L.: Solving the dna fragment assembly problem efficiently using iterative optimization with evolved hypermutations. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation. pp. 213–214. ACM (2010)
18. Maher, B., et al.: A firefly-inspired method for protein structure prediction in lattice models. Biomhc 4(1), pp. 56–75 (2014)
19. Mallén-Fullerton, G.M., Hughes, J.A., Houghten, S., Fernández-Anaya, G.: Benchmark datasets for the DNA fragment assembly problem. International Journal of Bio-Inspired Computation 5(6), pp. 384–394 (2013)
20. Minetti, G., Alba, E.: Metaheuristic assemblers of DNA strands: Noiseless and noisy cases. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC2010, Barcelona, Spain, 18-23 July 2010. pp. 1–8 (2010)
21. Minetti, G., Leguizamón, G., Alba, E.: Sax: a new and efficient assembler for solving dna fragment assembly problem. In: 2012 Argentine Symposium on Artificial Intelligence (2012)
22. Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H., Remington, K.A., et al.: A whole-genome assembly of drosophila. Science 287(5461), pp. 2196–2204 (2000)
23. Navarro, C.A., Hitschfeld-Kahler, N., Mateu, L.: A survey on parallel computing and its applications in data-parallel problems using gpu architectures. Communications in Computational Physics 15(2), pp. 285–329 (06 2015)
24. NVIDIA Corporation: NVIDIA CUDA C Programming Guide (June 2011)
25. Osaba, E., Yang, X.S., Diaz, F., Onieva, E., Masegosa, A.D., Perallos, A.: A discrete firefly algorithm to solve a rich vehicle routing problem modelling a newspaper distribution system with recycling policy. Soft Computing pp. 1–14 (2016)
26. Parsons, R., Forrest, S., Burks, C.: Genetic algorithms, operators, and DNA fragment assembly. Machine Learning 21(1-2), pp. 11–33 (1995)
27. de Paula, L., et al.: Parallelization of a modified firefly algorithm using GPU for variable selection in a multivariate calibration problem. International Journal of Natural Computing Research (IJNCR) 4(1), pp. 31–42 (2014)
28. Peters, H., Schulz-Hildebrandt, O., Luttenberger, N.: Fast in-place sorting with CUDA based on bitonic sort. LNCS, vol. 6067, pp. 403–410. Springer (2009)
29. Pevzner, P.: Computational Molecular Biology: An Algorithmic Approach. MIT Press (2000)
30. Pop, M.: Shotgun sequence assembly. Advances in Computers 60, pp. 193–248 (2004)
31. Saito, M., Matsumoto, M.: Variants of mersenne twister suitable for graphic processors. ACM Trans. Math. Softw. 39(2), pp. 12:1–12:20 (Feb 2013)
32. Sayadi, M.K., Hafezalkotob, A., Naini, S.G.J.: Firefly-inspired algorithm for discrete optimization problems: An application to manufacturing cell formation. Journal of Manufacturing Systems 32(1), pp. 78–84 (2013)
33. Shah, R., Narayanan, P., Kothapalli, K.: Gpu-accelerated genetic algorithms. In: Workshop on Parallel Architectures for Bio-ispired Algorithms in conunction with Parallel Architectures and Compilation Techniques (PACT Workshop). pp. 27–34 (2010)
34. Sutton, G.G., White, O., Adams, M.D., Kerlavage, A.R.: Tigr assembler: A new tool for assembling large shotgun sequencing projects. Genome Science and Technology 1(1), pp. 9–19 (1995)
35. Vidal, P., Olivera, A.C.: A parallel discrete firefly algorithm on gpu for permutation combinatorial optimization problems. In: Hernndez, G., Barrios Hernández, C.J., Díaz, G., García Garino, C., Nesmachnow, S., Pérez-Acle, T., Storti, M., Vázquez, M. (eds.) High Performance Computing, Communications in Computer and Information Science, vol. 485, pp. 191–205. Springer Berlin Heidelberg (2014)
36. Yang, X.S.: Nature-Inspired Metaheuristic Algorithms. Luniver Press (2008)
37. Yang, X.S.: Firefly algorithm, stochastic test functions and design optimisation. Int. J. Bio-Inspired Comput. 2(2), pp. 78–84 (Mar 2010)

38. Yang, X.S., He, X.: Firefly algorithm: Recent advances and applications. Int. J. Swarm Intelligence 1, pp. 36–50 (2013)
39. Yang, X.S., He, X.: Firefly algorithm: Recent advances and applications. CoRR abs/1308.3898 (2013)
40. Yang, X.S., Hosseini, S.S.S., Gandomi, A.H.: Firefly algorithm for solving non-convex economic dispatch problems with valve loading effect. Applied Soft Computing 12(3), pp. 1180–1186 (2012)

Pablo Javier Vidal, received his degree in Computing Engineering, in 2008 from Universidad Nacional de la Patagonia Austral; the degree fo Dr. in Software Engineering and Artificial Intelligence, in 2106, from Universidad de Málaga. He is currently an assistant professor at the Universidad de la Patagonia Austral-Unidad Académica Caleta Olivia. Currently has a postdoctoral scholarship from Consejo Nacional de Investigaciones Científicas y Técnicas. He has experience in software engineering with emphasis on automation and Graphic Processing Units research. His main research topics are parallel and distributed computing, evolutionary algorithms and metaheuristics, numerical modelling. ORCID: 0000-0001-6502-8010

Ana Carolina Olivera is an Assistant Researcher at National Council of Scientific and Technological Researches from the Ministerio de Ciencia y Tecnologa de la Nación Argentine. Dr. in Computer Science from Universidad Nacional del Sur, in Bahía Blanca, Argentina. She is Adjunct Professor at the Department of Exact and Natural Sciences of Universidad Nacional de la Patagonia Austral. She participates and directs in several national and international projects. Her research focuses on bio-inspired algorithms. She has published several book chapters and articles in indexed journals and proceedings of refereed international conferences. ORCID: 0000-0001-7825-1959

Received: May 10, 2017; Accepted: April 16, 2018.