Available online at www.sciencedirect.com
ScienceDirect
Computers & Operations Research 35 (2008) 2776–2790
computers & operations research
www.elsevier.com/locate/cor

DNA fragment assembly using a grid-based genetic algorithm☆

A.J. Nebro*, G. Luque, F. Luna, E. Alba

Departamento de Lenguajes y Ciencias de la Computación, E.T.S.I. Infomática, University of Málaga, Boulevard Louis Pasteur 29071, Spain

Available online 19 January 2007

Abstract

In this paper we propose a genetic algorithm (GA) for solving the DNA fragment assembly problem in a computational grid. The algorithm, which is named GrEA, is a steady-state GA which uses a panmitic population, and it is based on computing parallel function evaluations in an asynchronous way. We have implemented GrEA on top of the Condor system, and we have used it to solve the DNA assembly problem. This is an NP-hard combinatorial optimization problem which is growing in importance and complexity as more research centers become involved on sequencing new genomes. While previous works on this problem have usually faced 30 K base pairs (bps) long instances, we have tackled here a 77 K bps long one to show how a grid system can move research forward. After analyzing the basic grid algorithm, we have studied the use of an improvement method to still enhance its scalability. Then, by using a grid composed of up to 150 computers, we have achieved time reductions from tens of days down to a few hours, and we have obtained near optimal solutions when solving the 77 K bps long instance (773 fragments). We conclude that our proposal is a promising approach to take advantage of a grid system to solve large DNA fragment assembly problem instances and also to learn more about grid metaheuristics as a new class of algorithms for really challenging problems.

Keywords: DNA fragment assembly; Grid computing; Genetic algorithm

1. Introduction

Finding gene locations for specific functions is an important topic in bioinformatics research that requires accurate DNA sequences to be obtained. At present, DNA sequences that are longer than 600 base pairs (bps) cannot routinely be sequenced accurately. For example, human DNA is about 3.2 billion nucleotides in length and cannot be read at once. Hence, large strands of DNA need to be broken into small fragments for sequencing in a process called *shotgun sequencing*. In this approach, several copies of a portion of DNA are each broken into many segments short enough to be sequenced automatically by machine. But this process does not keep neither the ordering of the fragments nor the portion from which a particular fragment came. This leads to the DNA fragment assembly problem [1] in which these short sequences have to be then reassembled, in order, using the overlapping portions as landmarks. The automation allows shotgun sequencing to proceed far faster than traditional methods. But comparing all the tiny pieces and matching up the overlaps requires massive computation.

☆ This work has been partially funded by the Ministry of Science and Technology and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project).
* Corresponding author. Fax: +34 952 13 13 97.
E-mail addresses: antonio@lcc.uma.es (A.J. Nebro), gabriel@lcc.uma.es (G. Luque), flv@lcc.uma.es (F. Luna), eat@lcc.uma.es (E. Alba).

0305-0548/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2006.12.011

The assembly problem is therefore a combinatorial optimization problem that, even in the absence of noise, is NP-hard: given k fragments, there are $2^k k!$ possible combinations. Over the past decade, a number of fragment assembly packages have been developed and used to sequence different organisms. The most popular are PHRAP [2], TIGR assembler [3], STROLL [4], CAP3 [5], Celera assembler [6], and EULER [7]. Each one automates fragment assembly using a variety of algorithms, being the most widely used those based on greedy techniques. In this work, our interest is to apply genetic algorithms (GAs) [8] to tackle large instances of the DNA fragment assembly problem, since they are robust search methods requiring little information to search effectively in large or poorly understood search spaces and have been proven to outperform the greedy techniques on small instances of the problem [9]. GAs are members of a family of techniques known as evolutionary algorithms (EAs) [10] and they are said to be "evolutionary" because their basic behavior is drawn from Nature, and evolution is the basic natural force driving a population of tentative solutions towards those regions of the search space where the optimal solutions are located.

The working principle of a GA is to evolve from an initial population composed of tentative solutions applying stochastic operators such as selection, crossover or mutation, so that each new generation produces better individuals. The number of generations needed to reach a satisfactory solution is problem dependent. Also, there are many engineering and optimization problems in which function evaluation involves tasks demanding highly computational resources (e.g. real-world instances of the DNA fragment assembly problem). In this scenario, parallelism arises as an option for reducing this execution time down to affordable values [11]. In fact, GAs are naturally prone to parallelism due to their population-based approach in which each individual is an independent unit [12].

When dealing with populations of individuals, two parallelizing strategies are specially relevant: (1) parallelization of computation, in which the operations commonly applied to each individual (e.g. crossover, mutation, or evaluation) are performed in parallel, and (2) parallelization of population, in which the population is split into different parts, each one evolving in semiisolation (individuals can be exchanged between subpopulations). While the former strategy results in the master/slave model, the latter one leads to *distributed* (dGAs) and *cellular* (cGAs) models [11].

Therefore, if we consider large instances of the DNA fragment assembly problem, the evaluation of an individual can easily require more than several tens of seconds. Taking into account that it can be necessary to perform several hundred of thousands of individual evaluations, the total computing time can be in the order of hundreds of days. Such high-end computational resources cannot be addressed in normal clusters of machines. In this context, grid computing systems (or grids) [13] appear as a platform which provides the computing power of hundreds and thousands of computers, thus enabling to execute in a reasonable amount of time algorithms which otherwise would be considered as unfeasible.

In this paper we propose grid-based EA (GrEA), a parallel GA for solving the DNA fragment assembly problem. Our algorithm is based on the master/slave model, using a single population in the master and evaluating the individuals in parallel using the slaves. This model fits naturally in the working mechanisms of a GA, so it is becoming popular to develop GAs adapted to grid environments [14–18]. The idea under GrEA is to design a simple algorithm but powerful enough to adapt itself to the dynamic environment that constitutes a grid, with nodes frequently coming and going to and from the system. GrEA has been implemented on top of Condor [19], a system for constructing computational grids, using the master–worker (MW) library [20]. Our goal is to gain experiences using a small grid composed of up to 150 machines, in order to a further research in the line of solving hard instances of the DNA assembly problem with a larger system.

The contributions of the paper can be summarized in the following:

- We study and determine the requirements that a parallel GA for solving the DNA assembly problem should fulfill to be executed in a computational grid. As a result, we propose GrEA. The algorithm is implemented in C++ using the MW library on top of Condor.
- We study the behavior of GrEA when applied to solve a problem instance of 77,292 bps long (773 fragments). We propose and analyze two different approaches: firstly, we consider only the fitness function in the evaluation process; secondly, we study the influence of using improvement methods after the evaluation of the individuals with the goals of enhancing the search capabilities of the algorithm and the scalability of GrEA.
- We evaluate our proposal in a grid composed of up to 150 computers.

The rest of the paper is organized as follows. In Section 2, we analyze related works. In Section 3, we introduce the DNA fragment assembly problem. The GA for solving the DNA fragment assembly problem is described in

Section 4. Section 5 details GrEA, the grid-based GA approach. Some implementation details are given in next section. In Section 7 we present and analyze the experimental results. Finally, the conclusions and lines of future work are included in Section 8.

2. Related work

Nowadays, there is a number of groups working on using grid-enabled technologies for addressing bioinformatics problems, including the integrative genome annotation pipeline (iGAP), which has been used by the international consortium "Encyclopedia of Life" (http://eol.sdsc.edu/) for the extensive annotation of protein sequence data, myGrid (http://www.mygrid.org.uk/)—a large-scale grid-based European effort, North Carolina BioGRID (http://www.ncbiogrid.org/), EUROGRID (http://www.eurogrid.org/), and the Asia Pacific BioGrid Initiative (http://www.apbionet.org/apbiogrid/). Most of these bioinformatic projects are concerned with the handling of the huge amount of information coming from gene sequence data and data related to the physiology and biochemistry of organisms. This information must be integrated, analyzed, graphically displayed, and ultimately modeled computationally.

However, few proposals exist encompassing some kind of evolutionary algorithm and its implementation on grids in order to solve biotechnology problems. We analyze in this section those closely related to our work.

Imade et al. have presented in [16] the grid-oriented GA (GOGA) framework. It has been implemented in Java on top of Globus Tool Kit [21]. GOGA is aimed at supporting optimization problem solving in bioinformatics that are related to estimating models of biological experimentally observed phenomena and their computer simulation. These are difficult, multidimensional, multimodal, nonlinear problems which requires long time for evaluating a solution. As a consequence, the GA parallelization strategy used in the GOGA framework is the master/slave approach. Additional works involving GOGA are [22–24].

In [25], the minimization of the energy in protein tertiary structures is addressed with a new GA endowed with local search which has been implemented on the grid MP commercial middleware. The function to be optimized comes from the classical mechanics and consists of certain energy terms with force-field parameters.

To the best of our knowledge, our work is the first attempt to solve the DNA fragment assembly problem with a grid-enabled GA based on the Condor/MW framework.

3. The DNA fragment assembly problem

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. 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 Fig. 1 and they take place in the laboratory. After the fragment set is obtained, a 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 a coverage is satisfied. These steps are the last three ones in Fig. 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 lies 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, a 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 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. 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 highly coverage regions.

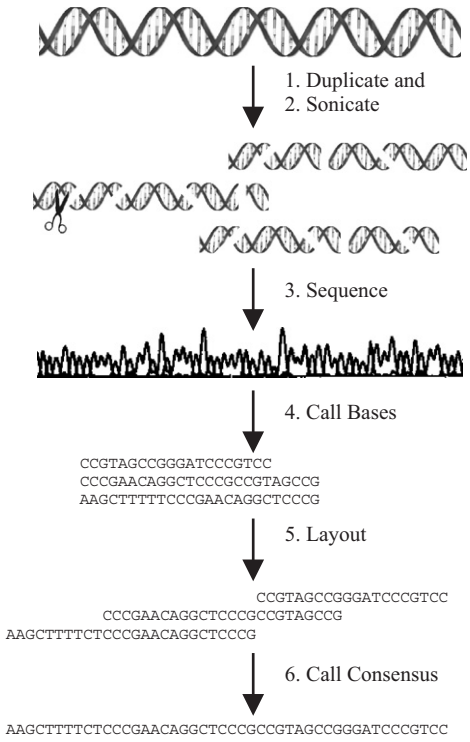

Fig. 1. Graphical representation of DNA sequencing and assembly.

(3) *Incomplete coverage:* It happens when the algorithm is not able to assemble a given set of fragments into a single contig. A contig is a sequence in which the overlap between adjacent fragments is greater or equal to a predefined threshold.
(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.

From a combinatorial optimization viewpoint, the whole process which results in the construction of the consensus sequence is similar to that of a tour in the traveling salesman problem (TSP). This is because each fragment would have to be in a specific fragment ordering sequence in order for the formation of a consensus sequence to take place. The main difference between TSP and DNA fragment assembly is that there would not be a proper alignment between the first and the last fragments in the consensus sequence that is comparable to the connection between the first and the last cities in the TSP solution. Therefore, many equivalent solutions to TSP are thus inequivalent in our context. Other important difference is that, while the ordering is the final solution to TSP, in our case, this ordering is only an intermediate step and several different orderings can produce equivalent results. Other minor differences can be found between both problems due to the challenges described previously (unknown orientation, incomplete coverage, etc.).

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, and 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 [1]:

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

where n is the number of fragments. The higher the coverage, the fewer the expected gaps, and the better the result.

4. A GA for solving the DNA fragment assembly problem

In this section we describe a GA for solving the DNA fragment assembly problem; this algorithm is the basis of GrEA. Firstly, we explain the working principles of a sequential GA; secondly, we define the solution representation and the genetic operators we have used; and, finally, we detail the fitness function.

4.1. Sequential GA

A GA is a randomized optimization procedure that uses information about the problem to guide the search (see Fig. 2). At each generation (iteration) t, a GA operates on a population of individuals $P(t)$, each one encoding a tentative solution, thus searching in many zones of the problem space at the same time. Each individual is a string of symbols encoding a solution for the problem and has an associated fitness value which is computed by the objective function. This fitness function is aimed at ranking the quality of the evaluated individual with respect to the rest of the population. The application of simple stochastic variation operators, such as mixing parts of two strings (*crossover*) or randomly changing their contents (*mutation*), leads this population towards fittest regions in an iterative manner. These operators are used with certain probability p_c and p_m, respectively. The algorithm finishes when a stopping condition is fulfilled (e.g., an optimum is found or a number of function evaluations has been carried out).

GAs can work in two basic ways. Firstly, the algorithm produces, from a population $P(t)$, a new population $P(t+1)$ with the new generated individuals, $P'(t)$, and probably the best individuals of $P(t)$ (*elitism*); it is said then that the GA is *generational*. Secondly, one or two individuals are created at every step and then they are inserted back into the population, consequently coexisting with their parents; this kind of GA is known as *steady state*

To improve the efficacy of a GA, an option is to apply an improvement method to an individual with the aim of accelerating the convergence towards the optimum. The mix of two different techniques in a unique algorithm is named *hybridization* [26].

The number of iterations required to obtain an acceptable solution (or the optimum) is related to the problem to be solved. In general, it is not unusual that a GA needs to perform tens or hundreds of thousands of function evaluations. In these cases, the computational cost of evaluating each individual has to be taken into account. For example, if we consider the instance of the DNA fragment assembly we have used here as working problem (see Section 7.1), this cost is around 15 s using a Pentium M 1.6 GHz processor running Suse Linux 10.0. This time does not seem to be very high, but if the algorithm has to carry out 500,000 function evaluations, the total computing time is in the order of 86 days, assuming that all the function evaluations require a constant amount of time. Furthermore, if we want to apply an improvement method after individual evaluation which would require, for example, 100 additional function evaluations, the entire task would need around 1700 s. So computing 500,000 of such tasks can be considered as intractable using

```
t := 0 ;
initialize & evaluate[P(t)] ;
while not stop_condition do ;
    P'(t) := variation [P(t)] ;
    evaluate [P'(t)] ;
    P(t+1) := select [P'(t) ∪ P(t) ] ;
    t := t + 1 ;
end while
```

Fig. 2. Pseudocode describing a sequential genetic algorithm.

a single computer. These arguments justify the need of using grid technologies for solving the DNA fragment assembly problem in a reasonable amount of time.

4.2. Solution representation and genetic operators

To represent an individual we have used integer permutations. A permutation encodes a sequence of fragment numbers, where consecutive fragments overlap. The solution in this representation requires a list of fragments assigned with a unique integer ID. For example, eight fragments would need eight identifiers: 0, 1, 2, 3, 4, 5, 6, and 7. This permutation representation requires special operators to make sure that we always get feasible solutions. Two conditions must be satisfied by every individual: all fragments must be presented in the ordering, and no duplicate fragments are allowed.

As to genetic operators, we use binary tournament as selection scheme. This operator works by randomly choosing two individuals from the population and the one having the best fitness is selected. The crossover operator we have considered is the order-based crossover (OX). This operator was specifically designed for tackling permutation problems. Given two parent individuals, the OX operator firstly copies the fragment IDs between two random positions in the first parent into the offspring's corresponding positions. Then, the rest of the fragments from the second parent is copied into the offspring in the relative order presented in this parent. If a fragment ID is already included in the offspring, it is skipped. Finally, the mutation operator is swap mutation, which randomly selects two positions from a permutation and it then swaps the two resulting fragments. Since no duplicated values are introduced, the mutated individual is always feasible.

4.3. Fitness function

In the DNA fragment assembly problem, the fitness function measures the multiple sequences alignment quality and finds the best scoring alignment. Several functions have been used in the literature [9]. For example, one of the most widely used is the following:

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

where w represents the overlap score between two fragments, and $f[i]$ is the ith fragment in the order. This function favors solutions in which strong overlaps occur between adjacent fragments in the layouts. But the actual objective is to obtain an order of the fragments that minimizes the number of contigs, being the optimal solution to reach one single contig, i.e., a complete DNA sequence composed of all the overlapping fragments. So the number of contigs is used as a high-level criterion to judge the whole quality of the results since it is difficult to capture the dynamics of the problem into a mathematical function. Contig values are computed by applying a final step of refinement with a greedy heuristic regularly used in this application [27]. We have found that in some (extreme) cases it is possible that a solution with a better fitness using F1 than other one generates a larger number of contigs (worse solution). Hence, we propose a variation of that equation that also takes into account an upper bound of the final number of contigs (c):

$$F(l) = F1(l) \cdot \left(1 + \frac{1}{c(l)}\right). \tag{3}$$

This new equation maximizes the overlap between adjacent fragments and, at the same time, minimizes the upper bound of the number of contigs. Computing the upper bound requires two steps. The first phase divides the solution into contigs. We iteratively check all fragments in the order creating a new contig when the overlap score between the current fragment and the next one in the order is less than the threshold. The second phase tries to merge the previous contigs into longer ones by processing all the combinations of pairs of contigs. Two contigs can be merged if the overlap score between the fragment in the end of the first contig and the fragment in the beginning of the second contig is greater or equal than the threshold. This phase finishes when all the combinations are processed or only one contig is left.

5. GrEA: a grid-based GA for solving the DNA fragment assembly problem

In this section we analyze firstly the requirements that a grid system imposes and how they can be dealt with a grid-enabled GA; then we present the algorithm GrEA and describe how it copes with those requirements.

5.1. Requirements imposed by the grid system

A grid system can be defined as a large collection of distributed resources connected by a network. In this context, we can distinguish two different software levels. In the top level we find the grid applications, which execute on top of the grid system; this software resides in the bottom level and it manages the underlying grid infrastructure and enables developing grid applications.

The resources in a grid typically share some of the following characteristics [28]:

(1) they are numerous,
(2) they are owned and managed by different organizations and individuals,
(3) they are potentially faulty,
(4) they can be added to the grid dynamically,
(5) they have different security requirements and policies,
(6) they are heterogeneous,
(7) they are connected by heterogeneous multilevel networks,
(8) they have different resource management policies, and
(9) they are likely to be geographically separated.

All these issues must be managed to some extent by the grid system software, while only some of them should be taken into account in the level of grid applications. We analyze next the elements that can influence the design of a GA for grids.

The fact of the resources being numerous (1) is the leitmotif of the grid systems, and it is the main reason why we discard the dGA and cGA models for our proposal of parallel GA. On the one hand, due that the resources are potentially faulty (3) and that new resources can be aggregated to the grid system (4), regular topologies such as rings, meshes, hypercubes, etc., are difficult to implement (i.e., the topology should be dynamically reconfigured at run time). On the other hand, the benefits of individual migration among subpopulations can be difficult to achieve in a system composed of thousands of nodes (e.g. the effect of migrating good individuals using a unidirectional ring topology may not affect to long distance subpopulations) in the case of dGA, and the ratio computation/communication can be unfavorable in the case of cGAs. These reasons lead us to consider the panmitic model in our proposal of grid-based GA.

A panmitic GA based on the master/slave model offers several advantages. Firstly, the model is conceptually simple: the master iteratively sends tasks involving the evaluation of individuals to the slaves, which respond with the fitness values; secondly, it requires the use of a star topology, which is simple to implement in a grid; finally, due to the stochastic nature of GAs, the working principles of the algorithm are not affected by the potential loss of a slave.

The successful implementation of a centralized scheme based on the panmitic model requires to cope with the following problems:

- Faults in the computer executing the master (3).
- Faults in a computer executing a slave (3).
- The algorithm should be aware of new processors dynamically added to the grid systems in order to use them as soon as possible (4).
- Different response time of the slaves because the processors in the grid can have different computing power (6) or delays due to the networks (7).
- Adjustment of the grain of the slave computations to avoid the master becoming a bottleneck (1).

These problems can be tackled at different levels: grid system software, grid-based GA, and problem to be solved. The different options are shown in Table 1. A choice between parenthesis indicates that it is a secondary option. For example, a crash in the machine executing the master can be handled by the grid system software if this provides

Table 1
Issues to consider by a grid-based panmitic GA and software levels where they are tackled

	Grid software	Grid GA	Problem
Master faults	Yes (No)	No (Yes)	No
Slave faults	No (Yes)	Yes (No)	No
New processors detection	Yes	Yes	No
Different response times	No	Yes	No
Slave computation grain	No	No	Yes

automatic checkpointing; otherwise, the grid GA should handle this issue. The same argument holds when a slave fails, although in this case we consider that the distributed GA can deal with this problem in several ways (requesting a new individual evaluation to another slave, or simply ignoring the problem). If new processors are incorporated to the system, the grid system software is responsible of providing a mechanism to notify the corresponding event to the algorithm, which should react consequently. The fact that the response time of the slaves is variable involves uniquely to the grid-based GA. Finally, the adjustment of the grain of the slave computation is directly dependent on the complexity of the fitness function evaluation, which depends on the problem at hand.

5.2. The algorithm GrEA

GrEA is a steady-state GA following the master/slave parallel model. The basic idea is that a master process executes the main loop of the algorithm and the slaves perform the function evaluations in an asynchronous way. Contrary to a sequential steady-state GA, in GrEA several evaluations are carried out in parallel; ideally, there should be as many parallel evaluations as available processors in the grid.

For better describing the algorithm, let us call GrEA-master the part of the algorithm corresponding to the master process, in opposite to the slave counterpart, named GrEA-slave. As commented before, GrEA-master executes the main loop as in a sequential GA but, whenever a new individual has to be evaluated, a task is created so that a slave process performs the computation. GrEA-master works in a reactive way: when an individual has been evaluated, it is inserted into the population and a step of the sequential GA is carried out, which leads to a new individual that will be sent to a slave to be evaluated. The mission of GrEA-slave is to receive an individual and to evaluate it; optionally, the individual can be improved by using an improvement method, so that the individual returned to GrEA-master can be different from the original one. We describe next the improvement method we use in GrEA.

The improvement method is based on using a simplified version of the evolutionary strategy (ES) method. In concrete, we use a $(\mu + \lambda)$ ES in which the mutation probability does not evolve with the individual, i.e., it is fixed. The overall method operates as follows: at each iteration, the ES procedure generates new λ individuals using a mutation operator starting from μ ones. Then, the best μ individuals taken from the μ old ones plus the newly generated λ ones are selected for the next iteration. This process is repeated until a termination condition is met. The mutation operator used for generating the new individuals is the *inversion mutation*. This operator randomly selects two positions from a permutation and then inverts the order of the fragment between these two fragment positions.

We analyze now how GrEA copes with the requirements imposed by the grid system. As discussed in the previous section, faults in GrEA-master are assumed to be solved by the grid system software; in the case of faults in processes running GrEA-slave, we adopt the approach of ignoring them, i.e., the individual assigned to that slave is lost (this does not affect the working principles of the GA). When a new processor is detected by GrEA, a GA step is executed and a new individual is generated for evaluation.

Assuming that the slaves simply evaluate individuals, it is obvious that faster processors will evaluate more individuals than slower ones due to that the response time of the processes running the GrEA-slaves code may vary. This introduces a light change in the "standard" behavior of the sequential GA because "old" individuals can arrive to the population when this has evolved several generations. Whether these individuals are discarded or they introduce significant perturbations in the population is a matter to be analyzed, but this is out of scope of this work. Another issue in this context is that we do not adjust computation grain of the slave: the time required to evaluate an individual depends on it and the power of the slave processor. In this situation, if the time required for an individual evaluation is small, an unbalance between computation/communication can occur.

We consider now the situation when we use the ES method. Here, the grain of a slave computation can be adjusted in several ways; for example, by setting different values of λ as well as modifying the number of iterations of ES. We take the approach of fixing the time a slave has to spend running the ES method. This way, both slow and fast processors will be computing a similar amount of time, so we can easily adjust the computation/communication ratio.

6. Implementation details

In this section we give a brief introduction to Condor as well as details about how we have used the MW library to implement GrEA.

6.1. Condor

Condor is a grid system software designed to manage distributed collections (pools) of processors spread among a campus or other organizations [19]. Each machine is supposed to have an owner, who can specify the conditions under which jobs are allowed to run; by default, a Condor job stops when a workstation owner begins using the computer. Hence, Condor jobs use processor cycles that otherwise would be wasted. Compared to other grid computing software, Condor is easy to install and to administrate, and existing programs do not need to be modified or re-compiled to be executed under Condor (they must only be re-linked with the Condor library).

Salient features of Condor includes remote system calls, job checkpointing and process migration. Furthermore, Condor pools can be composed of heterogeneous machines, and several pools can be combined using Globus [21] and Condor-G [29]. This way, Condor manages the issues characterizing a grid system which were commented above in Section 5.1.

6.2. The MW library

GrEA has been implemented using MW [20], a software library that enables to develop MW parallel applications on top of Condor using C++.

An MW application consists mainly of subclassing three base classes: MWTask, MWDriver, and MWWorker. A MWTask represents the unit of work to be computed by a slave. It includes the inputs and outputs to be marshaled to and from the slaves. In our implementation, the input is an integer array (the permutation representing an individual) and the output is a real value containing the fitness value plus another integer array (because the individual can be modified by an improvement method). The MWWorker provides the context for the task to run; in concrete, in GrEA the subclass of MWWorker contains the GrEA-slave code. Finally, the MWDriver subclass manages the whole process: creating tasks, receiving the results of the computations, and deciding when the computation is complete. The GrEA-master code is executed in this subclass.

The MW framework works with Condor to find computing resources for the available tasks, handle communication between the nodes, re-assign tasks if their current machine fails, and globally manage all the parallel computations. MW provides hooks to save the state of the driver, so that if the driver, or its machine crashes, the computation can make progress upon driver restart.

MW can run with one of several resource management and communication (RMComm) implementations. This layer implements communication between the master and the workers, and the management of the worker machines. There are several choices, including communicating via PVM, sockets, and shared files. We have chosen the last option because it is the most robust; for example, if the driver (master) crashes, the workers (slaves) can continue their computation, which is not possible using PVM and sockets. Although process communication using files is slow, our application is not intensive in data exchanges, and the communication costs can be acceptable if the computation time is long enough.

7. Experimental results

In this section we analyze the behavior of GrEA when it is executed in a grid system. Firstly, the target problem instance used is presented in Section 7.1. Then, the experiments performed are included in Sections 7.2 and 7.3.

Table 2
Parameter settings of the experiments

Parameter	Value
Population size	512 individuals
Representation	Permutation (773 integers)
Crossover operator	Order-based ($p_c = 0.8$)
Mutation operator	Swap ($p_m = 0.2$)
Selection method	Binary tournament
Replacement strategy	Worst individual

Our condor pool is composed of up to 150 computers belonging to several laboratories at the Computer Science Department of the University of Málaga. The pool includes UltraSPARC 477 Mhz processors running Solaris 2.8, and Intel and AMD processors (Pentium III, Pentium IV, Athlon) running at different speeds and executing several flavors of Linux (Suse 8.1, Suse 9.3, RedHat 7.1). We have used Condor 6.7.12, and MW 0.9. All the machines are interconnected through a 100 Mbps Fast Ethernet network.

The parameter settings of all the experiments are detailed in Table 2. Because of the stochastic nature of GAs, it is necessary to perform a number of independent runs (e.g. 30) of each test to gather statistically meaningful data. However, we have not been able to achieve a completely stable grid system to carry out such a number of tests for the many weeks they would require. For this reason, we describe two different kinds of experiments and analyze the results taking into account the data obtained when repeating them several times.

7.1. Target problem instance

A target sequence with accession number BX842596 (GI 38524243) was used in this work. It was obtained from the NCBI web site.[1] It is the sequence of a Neurospora crassa (common bread mold) BAC, and is 77,292 bps long. To test and analyze the performance of our algorithm, we generated a problem instance with GenFrag [30]. The problem instance, 842596_7, contains 773 fragments with average fragment length of 703 bps and coverage 7.

This instance is very hard due it is generated from very long sequence using a small/medium value of coverage and a very restrictive cutoff. The combination of these parameters produces a very complex instance. For example, longer target sequences have been solved in the literature [5], however, they used a higher coverage. The coverage measures the redundance of the data, and the higher coverage, the better the results can be found easily. The cutoff value is the minimum overlap score between two adjacent fragments required to join them in a single fragment. The cutoff, which we have set to 30 (a very high value), provides one filter for spurious overlaps introduced by experimental error. Instances with these features have been only solved adequately when target sequences vary from 20 to 50 k bps [9,27,31].

7.2. Experiment 1: 500,000 function evaluations

The first experiment consists of running GrEA to evaluate 500,000 individuals. We do not use the improvement method in the slaves, i.e., a task is just a function evaluation. We run the experiment twice, using two pool sizes. The obtained results are summarized in Table 3. The values included in the first four rows are reported by the MW library. Instead of using the speed-up to measure the parallel performance (we have not run a sequential version of the algorithm due to obvious reasons of problem complexity), we consider the time reduction, consisting of dividing the total CPU time consumed by the workers by the wall clock time, and the parallel efficiency, which is computing by dividing the time reduction by the average number of used workers.

From Table 3 we can observe that the parallel efficiency in both executions is 0.56. This relative low result is explained because of the grain of the worker computation, that is getting finer as the computation progress (we detail this issue below in this section). We should expect a better efficiency when solving a larger problem instance. Anyway, the time

[1] http://www.ncbi.nlm.nih.gov/.

Table 3
Results of experiment 1

	Execution 1	Execution 2
Total workers	67	124
Averaging used workers	65.5	109.7
Wall clock time	27.04 h	17.77 h
Total CPU consumed	40.28 days	45.3 days
Time reduction	35.7	61.2
Parallel efficiency	0.56	0.56
Best fitness found	250,808	247,361
Final number of contigs	2	2

reduction is important (taking the average of the two experiments, from 40 days to one single day) and justifies the use of GrEA on a grid system.

Considering the found solutions, they are in the same range of values. It is an expected result, since the algorithm performs the same number of function evaluations in both executions. Also, they obtain the same number of contigs. Although these solutions are not optimal (the optimum value is one contig), they are very accurate since it is far from trivial to compute solutions with a small number of contigs for real-world instances of this problem like this one.

We have analyzed the behavior of the algorithm observing the trace of the first execution. Concretely, we have measured the following issues at the beginning of each hour of computation time:

- The computing time required by a particular machine in the pool to evaluate the individuals it has been assigned to.
- The evolution of the best fitness value during the computation.
- The number of evaluated individuals per hour.

The obtained results are included in Fig. 3. We observe that the evaluation of the fitness function requires different computing times depending on the individuals (Fig. 3, top left), ranging in the beginning between 17 and 30 s. However, the evaluation time gradually decreases, being about 2 s in the last hours of computation and thus reducing the grain of the computation. The explanation of this behavior is related to the way the evaluation function works (see Section 4.3). The individuals in the early steps of GrEA have a large number of contigs, and the fitness function tries to reduce this number by executing an iterative greedy procedure. During the evolution process, the solutions tend to be more accurate, producing fewer contigs, so in the latter steps the fitness evaluation time is significantly reduced.

The curve tracking the fitness value per computation hour shows an almost linear shape (Fig. 3, top right). This suggests that the population has not converged yet, so we can expect to improve the solution of the problem if we increase the number of evaluations to be computed.

The number of evaluated individuals per hour (Fig.3, bottom) gradually increases, what is a consequence of the reduction of the computation time for evaluating individuals. Therefore, the number of evaluations can be considered as high in the last hours, when the evaluation times go down to around 1 s. This is an undesirable effect, because the ratio computation/communication is clearly unfavorable in this situation.

The conclusion of these analyses is that we have to adjust the evaluation time in the slaves in order to a better use of their computing power. We address this issue in the second experiment.

7.3. Experiment 2: using an improvement method in the slaves

From the previous experiment we deduce that it is necessary to increase the computation grain in the slaves when the search progresses. Here we analyze the use of the $(\mu + \lambda)$ ES described in Section 5.2. As commented in that section, the ES method is encompassed by an adaptive behavior to adjust itself its computation during a pre-fixed amount of time. The idea is to establish a timeout so that, when an individual is evaluated, we check if the timeout has elapsed; if not, the ES method is executed with the goal of improving the current individual. The ES main loop is run until the timeout has been consumed. At the end of the computation in the slave, the improved individual obtained by the ES algorithm is returned to the master. Therefore, tasks sent to the slaves may include right now several individual evaluations.

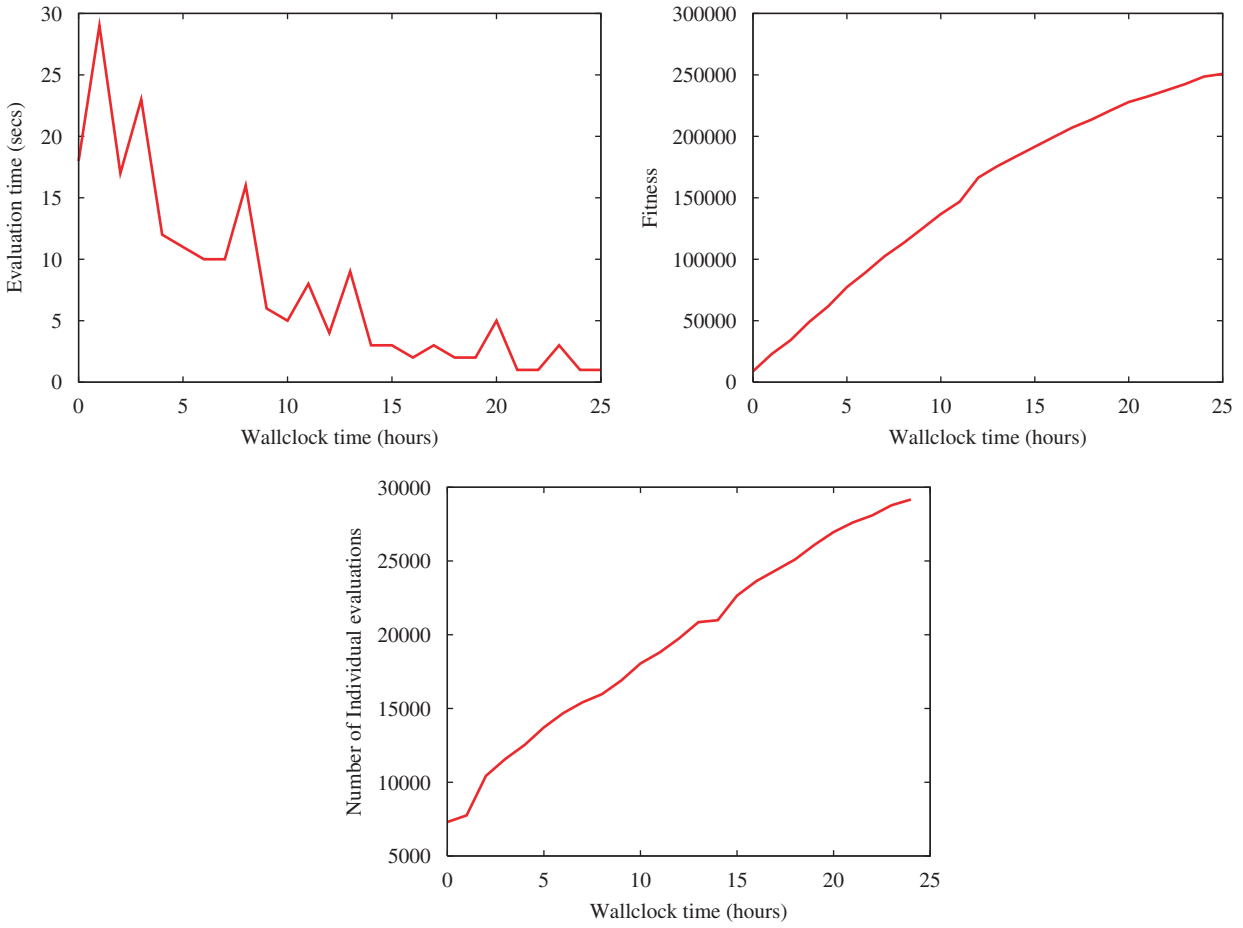

Fig. 3. Graphical results of analyzing the execution 1 of Experiment 1.

Once we have established the timeout, we ensure that all the slaves compute during a similar amount of time. As a consequence, the slaves in the fastest processors made a more intensive search than the ones executing in the slowest processors. Using this strategy, we intend to fix the problem of the unbalanced ratio computation/communication when the quality of the individuals is improved.

We have executed GrEA using the same parameter settings as the previous experiment (Table 2). The parameters defining the ES method are $\mu = 1$ and $\lambda = 10$, and the timeout in the slaves is set to 30 s. The stopping condition here is to evaluate 500,000 tasks, i.e., 500,000 individual evaluations plus the (1+10) ES. The results obtained are included in Table 4.

Compared to the previous experiment, we observe that the total CPU time consumed has increased to around 40–187 days, what is a logical consequence of using ES instead of merely evaluating the individuals. However, the parallel efficiency has grown up to 0.75 using more machines, thus indicating that the new approach improves the use of the CPU power of the grid resources. Although the number of contigs in both experiments is the same, the fitness value of the solutions found in this experiment improves previous results, indicating that solutions obtained in Experiment 2 are closer to the optimum. In fact, these results are quite remarkable since other approaches like CAP3 obtains solutions faster (less than 10 h) but the quality of the found solutions is worse (16 contigs).

In Fig. 4, top left we show the tracking of the individual evaluation in a particular computer of the grid at the beginning of each computation hour. We observe that in the first 10 h, the times oscillate between 30 and 59 s. This is an expected behavior, because we know from the previous experiment (see Fig. 3, top left) that an individual evaluation costs around 30 s at the beginning; thus, if this cost is less than 30 s (the timeout), at least one more evaluation is performed by the

Table 4
Results of experiment 2

	Execution 1	Execution 2
Total workers	149	146
Averaging used workers	138.4	138.2
Wall clock time	40.3 h	41.6 h
Total CPU consumed	187.71 days	191.4 days
Time reduction	111.8	110.5
Parallel efficiency	0.75	0.76
Best fitness found	289,077	287,200
Final number of contigs	2	2

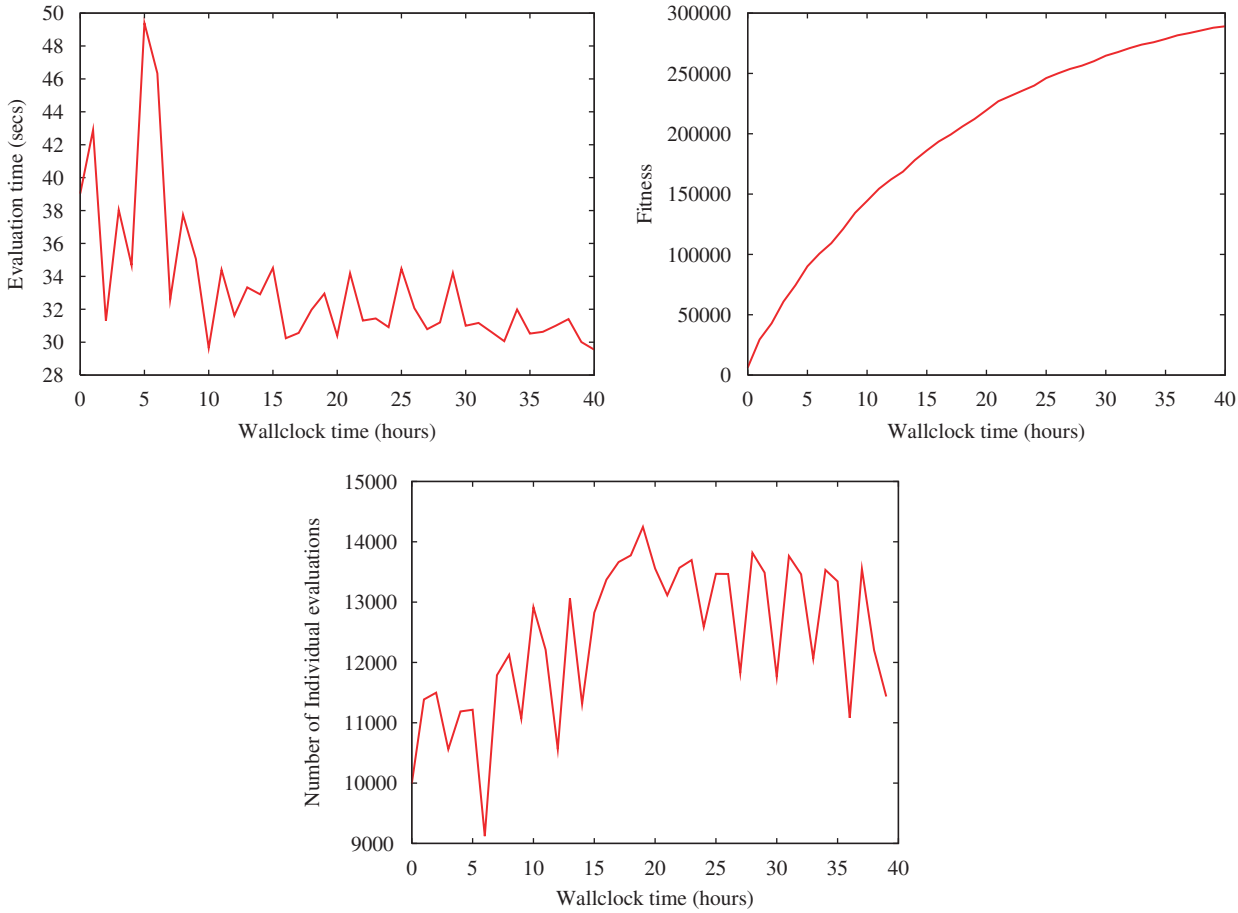

Fig. 4. Graphical results of analyzing the execution 1 of Experiment 2.

ES. Therefore, computing times just under 60 s are foreseeable. The individual evaluation times after the 10th hour oscillate between 30 and 34 s, what indicates that our approach allows the grain of the slave computations around to a prefixed value to be adjusted.

The shape of the curve tracking the fitness value per computation hour (Fig. 4, top right) leads us to think that the algorithm is near to converge to an optimum which is around the value 300,000. Given that we do not know the optimal solution to the instance problem being solved, more experiments should be carried out to assess whether we are near the optimal or a sub-optimal value.

The number of evaluated individuals per hour (Fig. 4, bottom) is kept among 9000 and 14,000 (Fig. 4 bottom). Compared to the previous experiment (see Fig. 3, bottom), in which that number increased linearly, we have achieved to keep the evaluated individuals per hour in a range in which the computation/communication ratio is favorable.

8. Conclusions and future work

We have solved the DNA fragment assembly problem using a grid-enabled GA which has been called GrEA. We have firstly analyzed the problem from the point of view of its solution by a sequential GA. Consequently, issues including the problem representation, genetic operators (selection, mutation, crossover), and fitness function have been addressed. In fact, we have proposed a new enhanced fitness function that takes into account an approximation of the number of contigs (the final goal of this problem). Later we have studied those requirements imposed by a grid system that must be taken into account by distributed GA targeted to that kind of system. The resulting algorithm is based on a panmitic parallel GA according to a master–slave scheme, and its main feature is that several individual evaluations are computed in parallel in an asynchronous way. This simple characteristic allows us to cope with the issues derived of executing the algorithm in a grid computing system, such as the presence of numerous processors, faults in some processors, the incorporation of new processors to the system, or different response times of the slaves because of differences in the power of the processors or delays due to the networks. GrEA has been implemented in C++ using the MW library on top of Condor. Our working grid has had a variable number of processors ranging from a few tens up to 150.

We have made two types of experiments in order to solve a 773 fragments instance of the DNA fragment assembly problem. Firstly, GrEA has been run to execute 500,000 function evaluations. The quantitative analysis of this experiment shows that a parallel efficiency of 0.56 can be achieved, what it is a remarkable result considering the fine granularity of the slave computations and that process communication is carried out via shared files. However, the analysis of the experiment reveals that the evaluation time of the fitness function decreases as the algorithm progresses, thus leading to a poor computation/communication ratio in the last hours of computation. Qualitatively, the results indicate that very accurate solutions to the instance problem can be obtained. The second experiment uses a simplified $(\mu + \lambda)$ ES with the aim of both dynamically adjusting the computation grain and enhancing the accuracy of the search. The obtained results show in this case that the parallel efficiency increases up to 0.75 using 149 processors while the best solutions outperforms those obtained by the first experiment.

Future research lines must count necessarily for a deeper study of GrEA dynamics and its usage in larger grid systems. This involves to solve more complex instances of the DNA fragment assembly problem as well as the analysis of using different improvement methods to obtain more accurate results.

References

[1] Setubal J, Meidanis J. Introduction to computational molecular biology. In: Fragment assembly of DNA. University of Campinas, Brazil; 1997. p. 105–39 [Chapter 4].
[2] Green P. Phrap, 2006, ⟨http://www.phrap.org/⟩.
[3] Sutton G, White O, Adams M, Kerlavage A. TIGR assembler: a new tool for assembling large shotgun sequencing projects. Genome Science & Technology 1995; 9–19.
[4] Chen T, Skiena S. Trie-based data structure for sequence assembly. In: The eighth symposium on combinatorial pattern matching; 1998. p. 206–23.
[5] Huang X, Madan A. CAP3: a DNA sequence assembly program. Genome Research 1999;9:868–77.
[6] Myers E. Towards simplifying and accurately formulating fragment assembly. Journal of Computational Biology 2000;2(2):275–90.
[7] Pevzner P. Computational molecular biology: an algorithmic approach. London: The MIT Press; 2000.
[8] Davis L. Handbook of genetic algorithms. New York: Van Nostrand Reinhold; 1991.
[9] Parsons R, Forrest S, Burks C. Genetic algorithms, operators, and DNA fragment assembly. Machine Learning 1995;21:11–33.
[10] Bäck T. Fogel DB, Mischalewicz Z, editors. Handbook of evolutionary computation. Oxford: Oxford University Press; 1997.
[11] Alba E, Tomassini M. Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation 2002;6(5):443–62.
[12] Alba E, Troya J. A survey of parallel distributed genetic algorithms. Complexity 1999;4(4):31–52.
[13] Berman F, Fox G, Hey A. Grid computing. Making the global infrastructure a reality, communications networking and distributed systems. New York: Wiley; 2003.
[14] Abdalhaq B, Cortés A, Margalef T, Luque E. Evolutionary optimization techniques on computational grids. In: ICCS 2002; 2002. p. 513–22.
[15] Jing T, Lim M, Ong Y. A parallel hybrid GA for combinatorial optimization using grid technology. In: IEEE congress on evolutionary computation; 2003. p. 1895–902.

[16] Imade H, Morishita R, Ono I, Ono N, Okamoto M. A framework of grid-oriented genetic algorithms for large-scale optimization in bioinformatics. In: Proceedings of the 2003 congress on evolutionary computation; 2003. p. 623–30.
[17] Eres M, Pound G, Jiao Z, Wason J, Xu F, Keane A, et al. Implementation and utilisation of a grid-enabled problem solving environment in Matlab. Future Generation Computer Systems 2005;21(6):920–9.
[18] Lo Presti G, Lo Re G, Storniolo P, Urso A, A grid enabled parallel hybrid genetic algorithm for SPN. In: ICCS 2004, Lecture notes in computer science, vol. 3036. Berlin: Springer; 2004. p. 156–63.
[19] Thain D, Tannenbaum T, Livny M. Condor and the grid. In: Berman F, Fox G, Hey T, editors. Grid computing: making the global infrastructure a reality. New York: Wiley; 2003. p. 299–335.
[20] Linderoth J, Kulkarni S, Goux J, Yoder M. An enabling framework for master–worker applications on the computational grid. In: Proceedings of the ninth IEEE symposium on high performance distributed computing (HPDC), Pittsburg, Pennsylvania; 2000. p. 43–50.
[21] Foster I, Kesselman K. Globus: a metacomputing infrastructure toolkit. International Journal of Supercomputer Applications 1997;11(2): 115–28.
[22] Imade H, Morishita R, Ono I, Okamoto M, A grid-oriented genetic algorithm for estimating genetic networks by S-systems. In: Proceedings of the SICE annual conference; 2003. p. 3317–22.
[23] Imade H, Morishita R, Ono I, Ono N. A grid-oriented genetic framework for bioinformatics. New Generation Computing 2004;22(2):177–86.
[24] Imade H, Mizuguchi H, Ono I, Ono N, Okamoto M, "Gridifying" an evolutionary algorithm for inference of genetic networks using the improved GOGA framework and its performance evaluation on OBI grid. In: Grid computing in life science: first international life science grid workshop, LSGRID 2004, Lecture notes in computer science, vol. 3370. Berlin: Springer; 2005. p. 171–86.
[25] Hanada Y, Hiroyasu T, Miki M, Okamoto Y, Mega process genetic algorithm using grid MP. In: LSGRID 2004, Lecture notes in bioinformatics, vol. 3370; 2005. p. 152–70.
[26] Talbi E-G. A taxonomy of hybrid metaheuristics. Journal of Heuristics 2002;8:807–19.
[27] Li L, Khuri S. A comparison of DNA fragment assembly algorithms. In: International conference on mathematics and engineering techniques in medicine and biological sciences; 2004. p. 329–35.
[28] Grimshaw A, Natrajan A, Humphrey M, Lewis M, Nguyen-Tuong A, Karpovich J, et al. From legion to avaki: the persistence of vision. In: Berman F, Fox G, Hey T, editors. Grid computing: making the global infrastructure a reality. New York: Wiley; 2003. p. 265–98.
[29] Frey J, Tannenbaum T, Foster I, Livny M, Tuecke S. Condor-G: a computation management agent for multi-institutional grids. In: Proceedings of the tenth IEEE symposium on high performance distributed computing (HPDC), San Francisco, California; 2001. p. 7–9.
[30] Engle M, Burks C. Artificially generated data sets for testing DNA fragment assembly algorithms. Genomics 16.
[31] Jing Y, Khuri S. Exact and heuristic algorithms for the DNA fragment assembly problem. In: Proceedings of the IEEE computer society bioinformatics conference. New York: Stanford University, IEEE Press; 2003. p. 581–2.