# The Genetic Algorithms Population Pluglet for the H2O Metacomputing System*

Tomasz Ampuła[1], Dawid Kurzyniec[1], Vaidy Sunderam[1], and Henryk Witek[2]

[1] Dept. of Math and Computer Science
Emory University, Atlanta, GA 30322, USA
{tamp,dawidk,vss}@mathcs.emory.edu
http://www.mathcs.emory.edu/dcl/
[2] Cherry L. Emerson Center of Scientific Computation and Dept. of Chemistry
Emory University, Atlanta, GA 30322, USA
hwitek@emory.edu

**Abstract.** This paper describes *GAPP* – a framework for the execution of distributed genetic algorithms (GAs) using the H2O metacomputing environment. GAs may be a viable solution technique to intractable problems; *GAPP* offers a distributed GA framework that can lead to rapid and efficient parallel execution of GAs from a variety of domains, with very little effort on behalf of the application scientist. It is premised upon the common phases embodied in GA lifecycles and contains modular implementations to handle each of them, whereas end applications simply provide domain-specific functions and parameters. *GAPP* is built for H2O, a component-oriented metacomputing system that enables cooperative resource sharing and flexible, reconfigurable concurrent computing on heterogeneous platforms. Experiences with the use of *GAPP* on H2O are described and preliminary results are very encouraging.

## 1 Introduction

Genetic algorithms (GAs) are known to be an effective approach for finding approximate solutions to complex (including NP-hard) problems. Often, when the theoretical nature of the problem is not well known, or no analytic algorithm is readily available, the use of nature-inspired techniques such as evolutionary algorithms or simulated annealing may be the only way to obtain near-optimal results in reasonable time [6].

Genetic algorithms are inspired by natural evolution, where stronger individuals in a population dominate, and eventually eliminate the weaker ones. The fitness of each individual depends on its unique set of chromosomes, i.e. a *genotype*. In the computational model, individuals represent candidate solutions to a given optimization problem. An individual genotype is represented by a data structure (a bit-vector in a canonical case) which is evaluated using a given fitness function. The evolution is simulated as an iterative process yielding a sequence of generations which are constructed by performing certain operations on individuals of the previous generation. The canonical set of operations include rating, selection, crossover, and mutations.

* Research supported in part by U.S. DoE grant DE-FG02-02ER25537 and NSF grant ACI-0220183.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3036, pp. 140–147, 2004.

For non-trivial problems, when individuals are complex or the computation of a fitness function is time-consuming, it is desirable to explore the potential for parallelism. There are four major models of parallel GA's [8,1]:

- *Single Population Farmer-Worker Model*: The farmer stores the whole population. Individuals are split into groups which are then sent to worker machines. Workers evaluate the fitness function and send results back to the farmer.
- *Fine Grained Single Population*: The single, large population is distributed among machines; however, the population has a spatial structure which limits interactions between individuals so that they can mate only with their neighbors.
- *Coarse Grained Parallel GA*: Many populations on many machines evolve independently, but individuals are allowed to migrate between populations (*The Island Model*)
- *Hybrid Parallel GA*: Various combinations of the above models.

In addition to the potential for speedup, the question of *qualitative* differences between distributed GAs and serial GAs has been studied in the past. A pioneering work involves Grosso's experiments of dividing a population into five demes [5], in which the rate of improvement was found to be faster in smaller demes than in one big population, and the rate depended on the model of migration between demes. Tanese proposed a 4D hypercube topology of subpopulations and reported that results as good as in a serial population model were found, with the added advantage of non-linear speedup [11]. An important factor in distributed GAs that may affect performance is the topology of the demes [2,4]. Tanese also found that migrating too many or too few individuals between subpopulations degrades performance. However good results were found faster that in serial GA even with no migration at all [12]. Other experiments showing differences between migration policies were also performed by Cantú-Paz [3]. It has also been shown that a parallel or distributed GA may outperform a serial GA in cases where partial solutions can be combined to form a better solution [9].

## 2 Design of the *GAPP* Framework

*GAPP* is a framework for coarse-grained distributed GA [8,1] computations and simulations on collections of heterogeneous machines. The framework has several objectives. First, it aims to facilitate the rapid development and deployment of distributed GA applications, and to harness the resources of multiple computer systems, regardless of their operating system or CPU speed. In addition, it is designed to enable seamless *resource sharing* and support ad-hoc collaborations. In this model, the end-user (a domain scientist) executes her GA application transparently on a dynamic, possibly large, and geographically distributed collection of machines to which access is supplied by independent contributors, without the need to have explicit login accounts established. Last but not least, the intention of the *GAPP* framework is to clearly separate aspects of GA application development, distributed data exchange, resource administration, and the actual end-user interface, so that the framework can be programmed easily and used by non-computer scientists. In order to support this model, four classes of actors in *GAPP* have been defined (while actual users may assume single or multiple roles), as illustrated in the Figure 1:

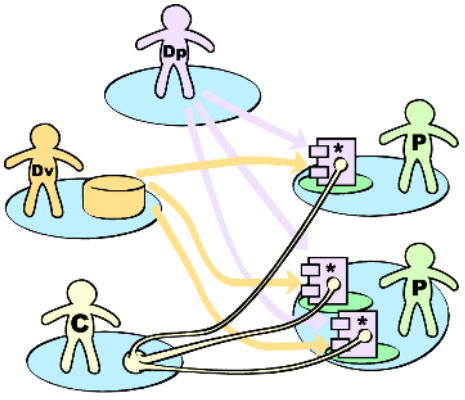


**Fig. 1.** Actors in *GAPP*: Providers (P), Deployer (Dp), Developer (Dv), Client (C), *GAPP* container(*) inside H2O kernel

- *Providers* (labelled **P** in the figure) supply computational power, by furnishing computer resources that host an environment suitable for running *GAPP* (although not necessarily exclusively). Providers define access privileges and grant security permissions to *deployers* (**Dp**) by specifying coarse- or fine-grained access policies (but need not grant login or other liberal access to their resources);
- *Deployers*: this group consists of users who dynamically install *GAPP* framework on the hosting environment supplied by providers, thus placing a layer over raw resources and enabling their use for GA computations;
- *Developers* (**Dv**)write the actual GA application code, which defines the fitness function and the population behavior with respect to standard GA operations. Importantly, the code is not concerned with aspects of population distribution, which are determined at run time. Once written, the code is published in a software repository (e.g. a Web server) to be available to *clients* (**C**);
- *Clients* are the end-users of *GAPP*. They harness resources supplied by providers, deployers and developers in order to solve their specific problems. Operationally, clients launch application codes (written by developers) within the *GAPP* framework (installed by deployers) on a collection of distributed resources (supplied by providers). These application codes operate on data specified by the client to solve a particular problem.

For example, in a GA approach to determine the shortest spanning tree in a graph, a developer implements an object that represents a spanning tree. The object must define methods required by *GAPP* to perform GA operations, such as crossover of two spanning trees. The deployer installs the *GAPP* container, and the client initializes it by providing the location of the application code in a software repository, and the data describing the graph in which the shortest spanning tree is to be found (e.g. a list of vertices). Note that end-clients do not have to have expertise in distributed GA programming. Having initialized sub-populations within *GAPP* containers, the client organizes them into a

desired topology and initiates the evolution process. The populations evolve independently; however, after each lifecycle, individuals migrate between populations. Every source population designates a specified number of its best-fit individuals and sends them to neighboring populations. Every destination population chooses, according to a specified policy, whether (or not) to accept incoming individuals and adds them to its list.

## 3 The H2O Hosting Environment

H2O is a lightweight, component-oriented framework for distributed computing [10]. It is based on a container-component model that readily hosts components written or wrapped in Java, and provides all of the necessary scaffolding and infrastructure for resource sharing, with high levels of security and control. Using this feature, users may avail of resources across multiple administrative domains without the need for login access or for cumbersome pre-arranged batch execution of distributed codes. H2O defines both an architecture and a methodology for the construction of application-specific or application-category-specific pluglets; *GAPP* follows this methodology that leverages the resource sharing framework support of H2O to facilitate distributed GA algorithm execution.

The distinction among actors in *GAPP*, presented in the previous Section, is in fact an instance of a more general model defined by H2O. In that model, *providers* supply computational resources equipped with *H2O kernels* (containers) and make them remotely available by defining appropriate access policies. *Developers* create code of the H2O-compatible components (so called *pluglets*) and distribute it or place it in software repositories. Subsequently, *deployers* instantiate these components within specific kernels. Finally, *clients* connect to those deployed pluglets and take advantage of services provided by them. One of the crucial, distinguishing features of the H2O container model is the separation of provider and deployer roles. This separation enables providers to share raw resources and to allow third-parties to provide added value (i.e. configure the raw resources) remotely by dynamically deploying or hot-swapping appropriate pluglets.

Due to the fact that the H2O design does not introduce global state (i.e. relationships between actors are defined as pairwise rather than group-oriented), the natural fault tolerance of the distributed island model can be leveraged. In an event of kernel crash, resource revocation, or a network partition, GA applications may continue to run (perhaps with small, but acceptable, decrease in efficiency) despite the loss of some subpopulations or connectivity between subpopulations.

## 4 Implementation of *GAPP*

The *GAPP* container has been implemented as a *population pluglet* – a generic GA-enabling component which may be plugged by deployers into H2O kernels. Once deployed, a population pluglet is responsible for instantiating, initializing and controlling populations, upon request from clients. At first, the client initializes his population by specifying its behavior (via a pointer to a code in a software repository) and initial data,

such as the population size, simulation starting point, etc. Then, the user may initiate the evolution process. Once the evolution has started, the client can stay connected and observe its progress, or may detach from the H2O kernel, and then reconnect and gather results at some later time. It is also possible to suspend and resume the running simulation, which may be useful for suppressing resource usage during peak hours.

To take advantage of the concurrent processing and distributed island model, it is necessary to interconnect population pluglets and to let individuals migrate between them. Every pluglet is connected to a generic *bridge* object, which is solely responsible for the distributed communication. That way, data exchange is abstracted away from the main course of the simulation, and does not have to be reimplemented within each GA code. Bridges between independent population pluglets (that usually run on distributed resources) may form one-one, one-many, or many-many, uni- or bidirectional connections. Therefore, any desired interconnect topology can be formed (see e.g. Figure 2).

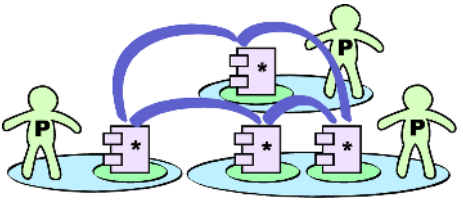


**Fig. 2.** An example of population pluglets connected into a ring.

The initial topology is specified by the client at startup time; several predefined canonical topologies include star, n-dimensional torus, and hypercube. Furthermore, the topology can be modified at run time: new islands can be added, existing ones removed, or the entire topology can be reconfigured e.g. in order to respond to varying network conditions or CPU loads, without disrupting the ongoing computations. During each lifecycle of the population, a fitness function is evaluated for every member of the population. Then, according to calculated fitness values, a ranking of individuals is created. Subsequently, the bridge is asked to copy a fixed number of the best-fit individuals and to distribute them among the bridges it is connected to. To increase efficiency and to avoid latency-related delays, communication is asynchronous so that the bridge does not wait for (or care about) delivery confirmation from other pluglets. Since migrations are not synchronized, faster machines are not constrained by slower ones, which gives users the freedom to configure topologies without regard to relative machine speeds.

## 5 Examples and Tests

*GAPP* has been applied in practice to solve several textbook-problems, as well as state-of-the-art research GA problems, and our results so far have been extremely positive. Brief descriptions are given below.

**Traveling Salesman Problem**: We applied the *GAPP* to solve TSP on a 380-node graph for which the optimal tour length is known to be 1621 units, according to an integer-rounded Euclidean distance norm [7]. The first example, illustrated in Figure 3, compares the performance of a single population of 100 individuals and a ring of 5 populations of 100 individuals each. As shown, the single population found a tour length of 1630, while in the same time the ring of 5 populations reached 1622, only a single unit worse than the optimal value.

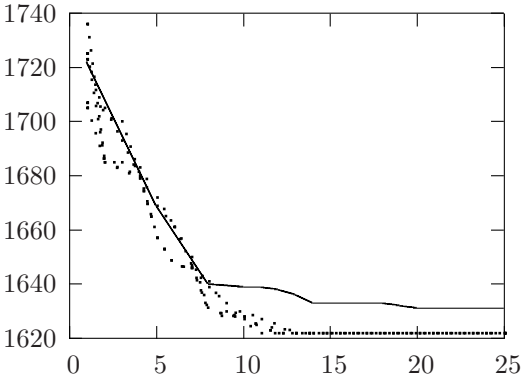


**Fig. 3.** Single population (solid line) versus five interconnected populations (dotted lines)

The second test compared the performance of a single large population (500 individuals) to five populations of 100 individuals each, again connected in a ring. Figure 4 shows that although both approaches yielded the same near-optimal value of 1622, it was reached by one of the ring members about 15 iterations sooner than by the single large population.

**Buckminster fullerene** ($C_{60}$) is a molecule consisting of 60 carbon atoms forming a truncated icosahedron. Among all possible systems consisting of 60 atoms of carbon distributed on a sphere, $C_{60}$ is well known to be a global minimizer of a potential energy. The optimization of potential energy of chemical structures is a difficult problem due to the fact that the number of local minima grows exponentially with the number of atoms, and because the function itself is expensive to calculate. We have applied *GAPP* to this problem, using a surrogate form of the actual potential [13] as a fitness function and starting from a set of randomly generated structures. Although the algorithm was not able to reproduce $C_{60}$ exactly, it obtained structures fairly close to it, as suggested by Figure 5. We are currently investigating the feasibility of using hybrid approaches

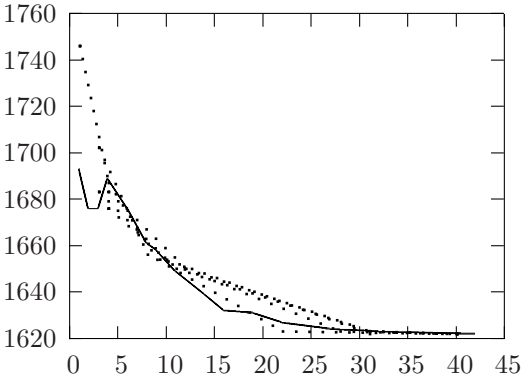


**Fig. 4.** Single large population (solid line) versus five small populations (dotted lines)

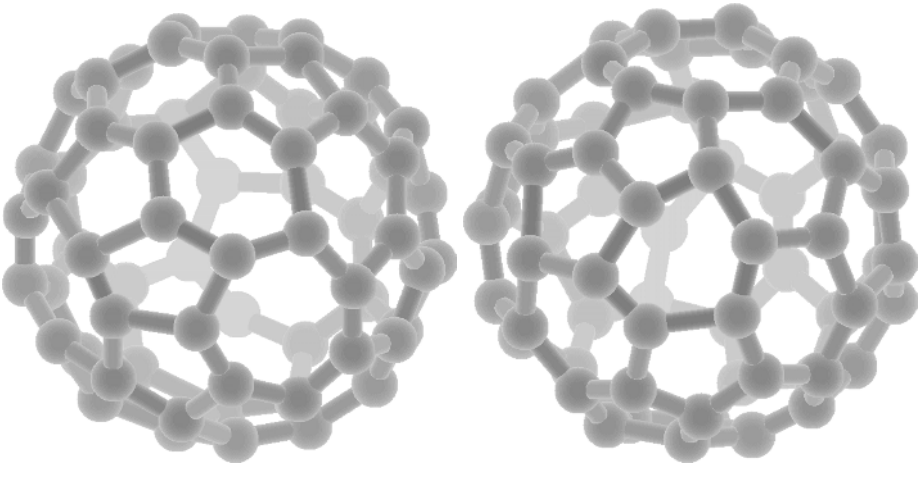

**Fig. 5.** Two examples of carbon nanoclusters generated by *GAPP*

(combining GAs with numerical optimization) to find exact global minima of potential energy.

The examples studied in this section indicate the potential of GA methods to find near-optimal solutions to NP-hard optimization problems commonly encountered in applied sciences. Also, they show that the distributed approach combined with the island model can improve performance and produce results much better than the local case. The natural scalability and fault tolerance of the island model are features which make it particularly appealing as a distributed computing application.

## 6 Conclusions

The *GAPP* framework offers an elegant solution to the parallel and distributed execution of GAs. Much of the details of distribution, managing populations in islands, and iterative evolution process is handled by the framework, thus necessitating only minimal, problem-specific effort on behalf of the domain scientist. Second, by leveraging the H2O system architecture, the role of developers and clients is decoupled, thus enabling the potential for increased exchange and cooperation among different groups of researchers who may leverage each other's efforts. Further, for applications dictated by the need for large computational resources, fault tolerance, steering, etc., H2O resource sharing across multiple administrative domains can be leveraged to significant advantage. These facilities and models for the execution of distributed GAs have been tested on several problems, and initial results are very encouraging. Hybrid techniques to combine GAs and traditional numerical methods, adjustments to improve the results in $C_{60}$ energy minimization, a better understanding of result-quality in the island model, and characterization of performance gains through parallel execution comprise the focus of current and future work on this project.

## References

1. T. C. Belding. The distributed genetic algorithms revisited.
2. E. Cantú-Paz. Topologies, migration rates, and multi-population parallel genetic algorithms.
3. E. Cantú-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms. *IlliGAL Report*, (99015), June 1999.
4. R. Gaioni and R. Davoli. Communication topologies for parallel genetic algorithms: A comparative study on cray t3d.
5. P. Grosso. Computer simulations of genetic adaptation: Parallel subcomponent interaction in a multilocus model, 1985.
6. W. Hart. A theoretical comparision of evolutionary algorithms and simulated annealing.
7. A. Rohe. http://www.math.princeton.edu/tsp/vlsi.
8. O. T. Sehitoglu. Gene reordering and concurrency in genetic algorithms.
9. T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H. Schwefel and R. Maenner, editors, *Parallel Problem Solving from Nature*, Berlin, Germany, 1991. Springer Verlag.
10. V. Sunderam and D. Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *The 11th International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.
11. R. Tanese. Parallel genetic algorithms for a hypercube. In *Proceedings of the Second Conference on Genetic Algorithms.*, 1987.
12. R. Tanese. Distributed genetic algorithms for function optimalization, 1989.
13. Y. Yamaguchi and S. Maruyama. A mollecular dynamics simulation of the fullerene formation process. *Chemical Physics Letters*, 286:336–342, April 1998.