ASYNCHRONOUS PEER-TO-PEER COMMUNICATION FOR FAILURE RESILIENT DISTRIBUTED GENETIC ALGORITHMS

José Carlos Clemente Litrán*
clemente@jaist.ac.jp

Xavier Défago†◇
defago@jaist.ac.jp

Kenji Satou*
ken@jaist.ac.jp

◇Graduate School of Information Science
*Graduate School of Knowledge Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan
†"Information and Systems", PRESTO
Japan Science and Technology Agency (JST)

ABSTRACT
This paper presents a grid service for solving optimization problems based on genetic algorithms. The proposed solution extensively uses peer-to-peer technology and epidemic protocols in order to improve scalability and failure resilience. This considerably relaxes the model traditionally used by genetic algorithm libraries. However, experimental results show that the convergence of the genetic algorithm is not necessarily impaired by the weaker model.

KEY WORDS
Genetic Algorithms, Fault Tolerance, Distributed Systems, Peer-to-peer

1 Introduction

Peer-to-peer systems and epidemic protocols constitute a powerful abstraction in large-scale distributed systems. In particular, these protocols are almost oblivious to changes in networking conditions, connectivity, and node failures. They are getting increasingly popular as the basis for e-business applications and similar data-intensive environments. However, because of their probabilistic nature, epidemic protocols are difficult to use in applications which require strong consistency and deterministic behavior.

Epidemic protocols ought to be used more often in computation-intensive environments, and in particular in Grid systems. In this paper, we present our progress on the development of a future grid service to support computations based on genetic algorithms. We plan to integrate this service in OBIGrid [1], a grid currently being deployed in Japan to support research in bioinformatics.

Genetic algorithms (GAs) constitute a popular optimization method for finding nearly optimal solutions to otherwise intractable problems. Roughly speaking, genetic algorithms manage a set of potential solutions (population), which are evaluated against a predefined problem-dependent fitness function. This population is then evolved applying to the individuals operations of crossover—pairs of solutions are combined randomly to form new ones—and selection—solutions with low fitness are discarded. After applying these operators repeatedly, the best solutions tend to converge toward an optimum.

There exist several libraries to support genetic algorithms in clusters. However, these platforms are often poorly scalable and severly lacking with respect to system fault-tolerance. By fault-tolerance, we mean that the computation is resilient to the failure of some system components. In particular, it is common for the whole computation to block as the result of a single processor crash. This is because many distributed GA libraries follow traditional implementations of genetic algorithms, which are based on a round-synchronous model. This constraint is however unnecessarily strong, and much can be gained by relaxing it. When relaxing the model, there is however a tradeoff between the robustness of the GA platform, and its ability to converge toward an optimal solution.

In this paper, we describe a GA platform based on a peer-to-peer model, and discuss its advantages and drawbacks. More specifically, we combine the so-called island model with epidemic communication. We present some experimental results showing that adopting a peer-to-peer model is not necessarily harmful to the convergence of the GA.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of genetic algorithms. Section 3 presents the system model. Section 4 describes the architecture of our platform. Section 5 evaluates our approach with respect to previous models. Finally, Section 6 discusses some open questions.

2 Background

2.1 Genetic Algorithms

GAs are an idealized computational model of Darwinian evolution based of the principles of genetic variation and natural selection [2, 3], commonly used for optimization problems. Candidate solutions are encoded as genes (usually in the form of bit strings). A population is a collection of genes, also called individuals. These are evaluated through a fitness function to determine how close they are to the optimum solution. The population is made to evolve based on the Darwinian principles, with fitter individuals (closer to the optimum solution) being selected to survive. The steps executed by GAs are as follows:

1. *Generate initial population.* A pool of individuals (hypothesis) is randomly created. These invididuals represent possible solutions to the problem we are trying to solve.

2. *Calculate fitness for each individual.* Each of the individuals is more or less close to an optimal solution according to a certain user-defined distance, named fitness function. The ones closer to the solution are considered to be better, in the sense they provide us with a better hypothesis to solve the problem.

3. *Select individuals for mating.* According to their fitness, good individuals are selected to mate with others. Hopefully, the new individuals to be created would be fitter, thanks to the good characteristics of their parents. This operation is called crossover.

4. *Select individuals for mutation.* Randomly, we choose some individuals from the pool to modify them, also in a random manner. This operation is called mutation, and has an vital role in GAs, as it assures convergence of the whole population given enough time.

5. *Create new generation.* With the new individuals created by crossover, mutation, and part of the previous population, we generate a new pool. This is called the next generation of the GAs. In this way, the population evolves on time gaps driven by means of natural selection.

6. *Check stop condition.* If we have not reached the ending condition (obtaining an individual with a certain fitness, having the average fitness of population over a threshold, having looped a number of generations, etc.) we go back to step 2. Otherwise, the GA run is finished.

2.2 Island Model

The island model is a popular way to implement distributed GAs [4, 5]. The basic idea is to set up populations of individuals to evolve independently in a set of *islands*. Periodically, individuals "sail" from one island to another, introducing genetical diversity in the arriving island. Following this metaphor, in this model each machine (island) executes a GA and maintains its own subpopulation for search. The machines work on consort by periodically exchanging a portion of their populations in a process called migration. The fact that every island evolves independently statistically guarantees that incoming individuals from other islands will in fact provide genetical diversity to the local population, thus helping to improve the global search. A total population N_{total} for a serial algorithm could be spread across M machines by giving each machine a subpopulation size of $N_{island} = N_{total}/M$.

Most island model implementations are based on a synchronous exchange of information. Every certain amount of generations, the islands send and receive a fixed number of individuals to and from other islands. The islands execute following the steps seen in Section 2.1, but they must synchronize regularly in order to exchange individuals, thus introducing waits in the system. This model, inherited from parallel machines, is poorly suited to distributed systems. For instance, it does not properly support heterogeneity of the nodes, since the speed of the whole system is limited by the slowest node (as fast machines do have to wait for slower ones in order to synchronize). Communication delays are unpredictable, introducing wait states in the system due to the need for synchronization. Finally, failures (loss of messages or crashing nodes) are not handled at all and provoke the complete stop of the system.

3 System Model

In this paper, we assume a distributed system composed of a large number of nodes, which exchange information by message-passing only. Every node has its own unique identifier. Communication is assumed to be subject to message loss and network partitions, both of which of a transient nature. Nodes can fail by crashing and we assume that at least one node remains operational.

We assume the availability of some peer-to-peer group membership service, such as SCAMP [6]. In SCAMP, group members do not usually have a global knowledge of the composition of the system, but rather they have only a partial view of it. SCAMP ensures, with very high probability, that (1) the global view is given by the aggregation of all local views, and (2) the size of local views grows logarithmically with respect to the size of the whole system. Nodes exchange information by gossiping with their direct neighbors (i.e., the members of their local view).

SCAMP supports three basic operations: subscription, unsubscription and recovery from isolation. New nodes can join the system by sending a subscription request to an arbitrary member, and start with a local view consisting of the member they sent the request to. Any node receiving such a request will send the new node-id to all the members of its local view, and create a certain number of copies of the new subscription, forwarding them to randomly chosen members in its local view. When receiving a forwarded subscription, a node can integrate the new subscriber to its view or forward the subscription to a randomly chosen node from its local view.

Nodes receiving unsubscription request will delete the unsubscribing node, given that it is on the node's local view. Isolated nodes (those whose identifier is not present in any node's local view) can recover by sending a request to an arbitrary node of its partial view if they have not received any messages for a given period.

4 Architecture

In order to construct our platform, we start by building up our own island model GA. Using a non-distributed GA package GAJIT [7], we include a new operator called *migration*. The function of this operator is to probabilistically send individuals from the local population to remote islands. Also, every island should check for incoming individuals coming from other nodes. The steps presented in Section 2.1 are modified as follows:

4b. *Select individuals for migration.* According to their fitness, copies of certain individuals are chosen to migrate to other islands. By selecting copies of individuals we guarantee constant populations on every island, which is desirable in order to compare with traditional models.

5. *Create new generation.* If there are any incoming individuals from other islands, incorporate them to the new local population on a probabilistic basis. The rest of the new generation is created using crossover and mutation operators, and by passing individuals from the last generation.

Asynchrony of the system is achieved by making a non-blocking call in step 5: if there are no incoming individuals in the local queue, we proceed to the next step. In this simple way, we remove the need for synchrony in the system and achieve fault tolerance to crashing members or lost messages de facto.

The architecture of a traditional distributed GA platform is depicted in Fig. 1. Issues as fault-tolerance or scalability are usually not addressed. Communication among nodes is synchronous, every node has a global view of the system and the number of members is small.

Our platform, on the other hand, is aimed at a distributed system where fault-tolerance is an important issue, since messages can be lost and nodes can crash. Communication is then asynchronous, nodes have partial views of the system and the number of members is large (see Fig. 2).

In order to study more easily the impact of migrations in the system, we implemented a migration policy system that allowed us to control several related parameters:

1. *Which individuals to send.* As with other genetical operators, individuals selected to migrate are usually chosen probabilistically, so we benefit the ones with better fitness.

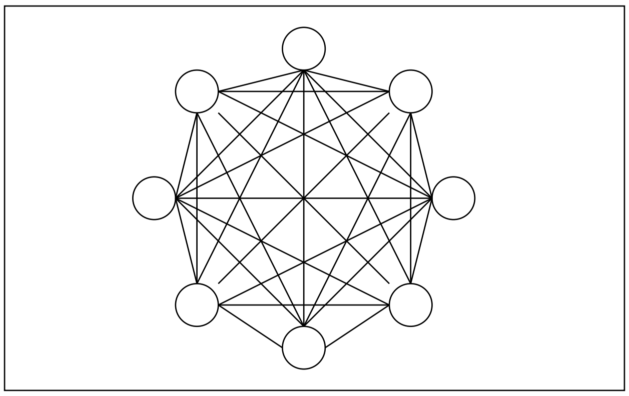
Figure 1. Traditional distributed GA: global view, synchronous communication, small number of islands

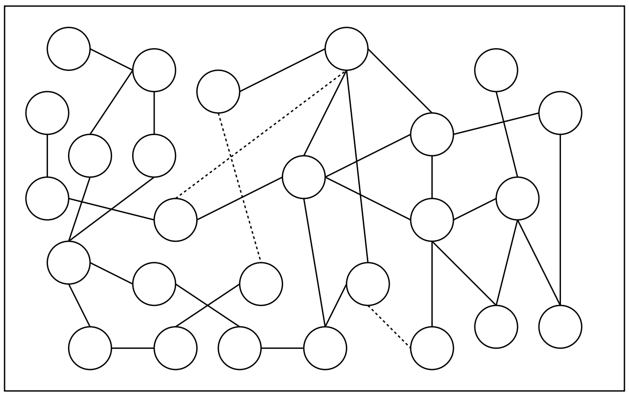
Figure 2. New approach for distributed GA: partial view, asynchronous communication, large number of islands

2. *When to send.* The interval among migrations is usually selected in number of generations. If we send too often, the whole population tends to become uniform, losing therefore the advantage of a distributed search in the solution space. On the other hand, if the gap among migrations is too big, islands tend to converge independently thus losing the property of collective search.

3. *Where to send.* As with the previous one, this parameter has a direct effect on the global convergence, for the same reasons stated above. These two parameters present also some side effects. First, the more often and the more islands we send to, the more we will increase the network traffic. More importantly, we suspect there could be a trade-off between convergence and resilient to fault tolerance (see Section 6 for more details).

4. *Send copy or original.* Sending *original* individuals would make the local population to decrease, while sending copies makes populations stable.

5. *Acceptance of incoming individuals.* We can either

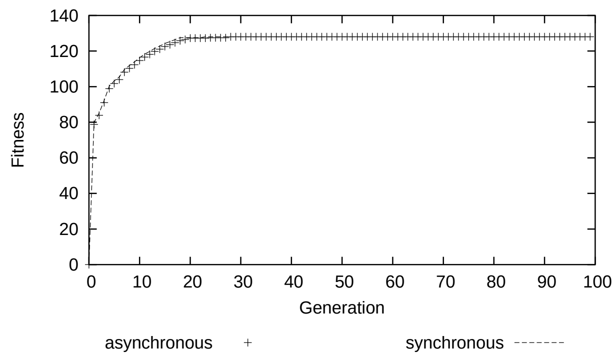

Figure 3. Synchronous vs asynchronous GA: same machine speed

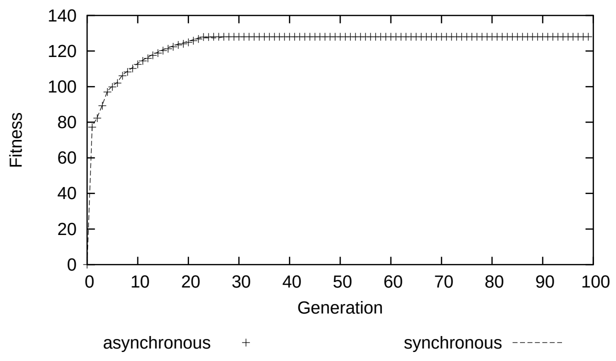

Figure 4. Synchronous vs asynchronous GA: different machine speed

choose to accept any incoming individual or select them probabilistically. Combined with the previous parameter, this could lead to islands gradually losing individuals or gaining them.

We want our system to be resilient not just to member crashes, but also to loss of messages. Therefore, we decided that individuals to migrate should be converted to bit streams and then sent as UDP datagrams. Since their arrival cannot be guaranteed, this effectively simulates random loss of messages in the network.

5 Evaluation

When building up the system, our first concern was how the distributed environment affects the convergence of the GA. Previous studies (as for instance in Whitly et al. [5]) show how the island model in fact helps improving the overall convergence of the system (given that the complexity of the problem we are solving is above a specific, problem-dependant minimum threshold). The use of asynchronous communications to ensure fault-tolerance does also not affect convergence, and in some cases can even have a positive effect on both convergence and performance.

We tested these two statements with our own system. As a first step, we compared results between non-distributed and distributed GAs, and as expected the distributed version outperformed the non-distributed one as long as the complexity of the problem was above a certain threshold. This result empirically proves the first statement, i.e., island model can help to improve the convergence of the system.

We set up then two experiments to compare synchronous versus asynchronous versions of our system. In the first one, all islands had the same configuration and characteristics (memory, speed, etc.). The results show very similar behaviour for both models (see Fig. 3).

In our next experiment, islands had different speed. We wanted to test how does synchrony and asynchrony behave when we introduce waits in the system. In terms of convergence, the results were very similar (see Fig. 4). Nevertheless, the average time needed to obtain a "perfect" individual was reduced by 4% when using the asynchronous model. Although we could not test this experiment with extremely large number of islands, we suspect the more the members, the higher the difference in time between synchronous and asynchronous systems. A full discussion of these results can be found in our previous work [8].

It is important to notice how although asynchronous GA models have been proposed, none of them considered fault-tolerance issues since they were aiming at small groups of islands. As we intend to build a platform with a large number of members, the need for a peer-to-peer, gossip-based system arises as a very reasonable approach.

The advantage of this kind of system is then based on the fact that, while achieving similar performance as traditional systems, it supports fault tolerance (islands can be lost, messages can be lost, but the whole computation is not lost) and heterogeneity (faster machines do not have to wait for slower ones).

6 Future Work

There are some questions that are still open in our platform. For instance, in non-distributed GAs, an average fitness function is usually implemented to measure the progress of the execution. In distributed systems, the problem of calculating global aggregate functions over the votes of individual group members has already been addressed in the literature. We will use the approach presented by Gupta et al. [9], since it allows message and member failures (which is not addressed in the work by Garg [10], for instance), and it does not depend on a fixed network topology (as in the work by van Renesse [11]). Roughly speaking, the idea is to construct an abstract hierarchy and then use gossiping within this hierarchy to evaluate the function for the whole group. This approach, while being robust to random message losses and crashes of group members, still guarantees a good probability of a member vote being included in the global estimate.

In our platform, crashed members do not have any effect in other islands. Individuals sent to them will be lost, and since there is no need to wait for synchronization, there is no further impact on the whole system. It could be desirable though, to be able to detect when a node has actually crashed. To address this problem, we intend to incorporate ideas proposed by Fetzer [12]

The way of gossiping messages among members is also of special interest in our system. In terms of convergence, if we do not send messages (individuals) to enough islands, our system would turn into a collection of independent members with no real distributed work towards a better global solution. On the other hand, sending to too many of them results in a slower convergence (as the global population tends to become too homogeneous). As seen in Section 3, we will use SCAMP membership service, where nodes just have a partial view of the system. We suspect there could be a trade-off between convergence and resilient to fault-tolerance depending on to how many members we choose to send (i.e. size of the partial view), but our results are not enough to establish a solid conclusion on this point yet.

7 Conclusion

In this paper we presented a platform for distributed, asynchronous, fault-tolerant GA. Fault tolerance and scalability issues have not been addressed in GA systems previously, since most of them are aimed at small groups of members. Our results show how asynchrony provides a basic but powerful mechanism to achieve fault tolerance. We also discussed on the importance of the relation among convergence, scalability and resilience to failures, which still has to be studied in depth.

8 Acknowledgments

We would like to thank Prof. Akihiko Konagaya and Prof. Isao Ono for their help and advice about this work. This work was supported by Grant-in-Aid for Scientific Research on Priority Areas (C) “Genome Information Science” from the Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

[1] Akihiko Konagaya. OBIgrid: Towards a new distributed platform for bioinformatics (invited paper). In *Proc. 1st Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS), 21st IEEE Int’l Symp. on Reliable Distributed Systems (SRDS-21)*, pages 380–381, Osaka, Japan, October 2002.

[2] John H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[3] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[4] Reiko Tanese. Distributed genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–439, Madison, USA, 1989.

[5] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. Exploiting separability in search: The island model genetic algorithm. *Journal of Computing and Information Technology*, 7:33–47, 1998.

[6] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the 3rd International Workshop on Networked Group Communication*, London, UK, 2001.

[7] Matthew Faupel. Gajit: A simple java genetic algorithms package, 1998.

[8] José Carlos Clemente Litrán. Distributed genetic algorithms: an asynchronous, fault-tolerant approach. Technical report, Japan Advance Institute of Science and Technology, 2003.

[9] Indranil Gupta, Robbert van Renesse, and Kenneth P. Birman. Scalable fault-tolerant aggregation in large process groups. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 433–442, Goteborg, Sweden, 2001.

[10] Vilay K. Garg. Methods for observing global properties in distributed systems. *IEEE Concurrency*, 5(4):69–77, 1997.

[11] Robbert van Renesse. Scalable management with astrolabe. Technical report, Dept. of Computer Science, Cornell University, 2000.

[12] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, 2003.