Distributed Beagle: An Environment For Parallel And Distributed Evolutionary Computations

Christian Gagné, Marc Parizeau, and Marc Dubreuil [a]

[a] Laboratoire de Vision et Systèmes Numériques (LVSN),
Département de Génie Électrique et de Génie Informatique, Université Laval, Québec (QC), Canada, G1K 7P4.
{cgagne,parizeau,dubreuil}@gel.ulaval.ca

Evolutionary computation is a promising artificial intelligence field involving the simulation of natural evolution to solve problems. Given its implicit parallelism and high computational requirements, evolutionary computation is the perfect candidate for high performance parallel computers. This paper presents Distributed BEAGLE, a new master-slave architecture for parallel and distributed evolutionary computations. It is designed as a robust, adaptive, and scalable system targeted for local networks of workstations and Beowulf clusters. Results obtained with a plausible deployment scenario demonstrate that system performance degrades gracefully when failures occurred, while still achieving near linear speedup in the ideal case.

Le calcul évolutionnaire est un champ prometteur de l'intelligence artificielle impliquant la simulation de l'évolution naturelle afin de résoudre divers types de problémes. Etant donné son parallélisme intrinséque et ses forts besoins en puissance de calcul, Il est le parfait candidat pour les ordinateurs paralleles de calcul haute performance. Cet article présente "Distributed BEAGLE", une nouvelle architecture maître-esclave pour les calculs évolutionnaires paralléles et distribués. Celle ci est conçue pour etre robuste, adaptative, extensible et adaptée à des réseaux locaux de stations de travail de même qu'aux grappes de calcul Beowulf. Les résultats obtenus avec un scénario possible de déploiement démontrent que les performances du système se dégradent lors de pannes alors que dans le cas idéal, le gain de performance est linéaire.

1 Introduction

Evolutionary Computations (EC) [1] is a promising machine intelligence discipline involving the simulation of natural evolution on computers. It is a generic problem solving method applicable whenever solutions can be represented by some data structure and evaluated by an objective function; the so-called "fitness" function. Populations of solutions – initially random solutions – evolve over time through a sequence of processes that include (natural) selection and different genetic operations. In the end, the fittest individual is chosen as "the" solution to the problem and, although EC systems do not in general guaranty convergence to an optimal solution, they have been shown in practice to outperform other techniques as well as human experts for several hard problems [2,3].

However, this generic capacity often comes at a high computational cost, especially when the evaluation of the fitness function is time consuming, which is usually the case for non trivial problems. On the other hand, the evolution process is implicitly parallel as every individuals composing a population of potential solutions can be processed mostly independently. This paper introduces Distributed BEAGLE, an extension of the Open BEAGLE EC framework [4], where evolutionary processes can be efficiently and easily distributed on a network of loosely coupled computers.

Paper structure goes as follows. A presentation of the principal EC flavors is first conducted. Then some nomenclature is presented about the four main types of Parallel and Distributed Evolutionary Computations (PDEC). Thereafter, Open BEAGLE is summarized before introducing the *distributed* BEAGLE architecture. The paper finishes with an analysis of the merits and limitations of the proposed system.

2 Evolutionary Computations

A specific instance of an EC, an Evolutionary Algorithm (EA), can be seen as an optimization process in which a population of solutions evolves over time, to solve a given problem. The EC principles have been successfully applied to numerous situations where classical methods of optimization, classification and automatic design were not able to produce adequate solutions. EC is generally divided in four major flavors: genetic algorithms (GA), genetic programming (GP), evolution strategy (ES), and evolutionary programming (EP).

Genetic algorithms (GA) [5,6] involve the evolution of a population of individuals representing possible solutions to a problem. An individual is usually a string of symbols defined over a given alphabet, most often bits (individual = string of bits). The idea is inspired from genetic DNA that composes every living creature. The search process is made by an iterative application of genetic operators, such as crossover, mutation, and Darwinian selection operators biased toward the fittest individuals. Using this Darwinian paradigm, a population of solutions evolves until some stopping criterion is reached. Figure 1 depicts a flowchart of the general GA algorithm. GA has been widely used as a numerical method to optimize parameters of systems without having any *a priori* knowledge about the search space. Only a feedback criterion is needed to give an objective value that guides the search.

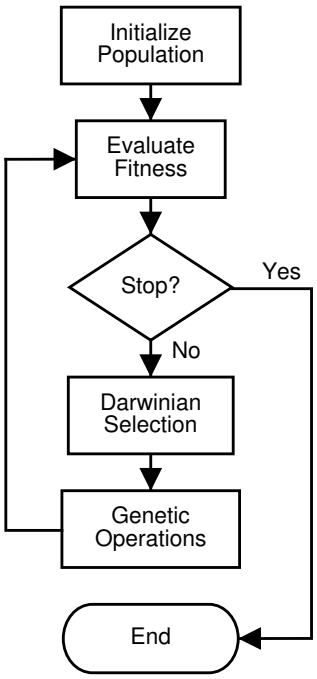

Figure 1. Classic GA Flowchart.

Genetic programming (GP) [7,8] is a paradigm that allows automatic programming of computers by heuristics inspired from the same evolution principles as GA: genetic operations of crossover and mutation, and natural selection operation. The difference between GA and GP lies mainly in the representation used, for which GP is similar to a computer program structure. Canonical GP was first formally expressed by Koza in the beginning of the 1990s [7,9]. Koza's GP represents programs as trees, that is acyclic and undirected graphs, where each node is associated to an elementary operation specific to the problem domain. Others have experimented with different representations, such as linear programs [8] or cyclic graphs [10]. GP is particularly adapted to the evolution of variable length structures.

The evolution strategy (ES) paradigm was developed by I. Rechenberg and H.-P. Schwefel at the Berlin Technical University in the 1960s [1,11]. In ES, each individual is a set of characteristics of a potential solution. This set is generally represented as floating-point vectors of fixed length. ES is applied to a parent population (of size μ) from which individuals are randomly selected to generate an offspring population (of size $\lambda \gg \mu$). Offsprings are modified by mutation, which consists in adding a randomly generated value that follows some parametrized probability density function. The parameters of this probability density function, called the *strategy parameters*, themselves evolve over time following the same principles. To engender a new population of size μ, the best μ individuals are chosen within either the λ offsprings (approach (μ, λ)), or the μ parents and λ offsprings (approach $(\mu + \lambda)$). Modern ES can also be nested.

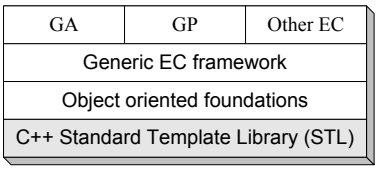

Figure 2. Open BEAGLE Framework Architecture.

Evolutionary programming (EP) has been developed by L.J. Fogel in the 1960s and later by D.B. Fogel et al. in the 1990s [12,1]. EP was initially designed to evolve finite state machines and has been later extended to parameter optimization problems. The approach is more focused on the relation between parents and offsprings than on the simulation of nature-inspired genetic operators. Contrary to the three first EC flavors, EP doesn't involve the use of a particular representation, but rather a high-level evolutionary model and a representation appropriate for the problem to solve. Only a mutation operator specific to the representation is needed. To do EP, a population of μ solutions is randomly generated. Each individual of the population produces λ offsprings resulting from mutation. Then, a natural selection operation is applied to produce a new population of μ individuals. The mutation - selection process is applied iteratively until a good solution is found.

3 Open Beagle

Open BEAGLE[1] [4] is a C++ framework for doing almost any kind of EC. Its architecture follows the principles of Object Oriented (OO) programming, where some abstractions are represented by loosely coupled objects and where it is common and easy to reuse code. Open BEAGLE has a three level architecture as illustrated in Figure 2. The OO foundations are the basis of this architecture, as a high-level extension of C++, inspired by *design patterns* [13,14]. It offers basic functionalities like smart pointers and garbage collection, object allocators, standard containers, and XML readers/writers. The generic EC framework implements basic mechanisms and structures to design versatile specialized Evolutionary Algorithms (EA). It is summarized in Figure 3.

The generic EC framework comprises three main components: a vivarium, an evolution system, and an evolver. The vivarium is a container for demes of generic individuals. The individuals themselves are specified by an abstract genotype. This genotype can be instantiated to any relevant structure (in Figure 3, it is shown as a GP tree, but this is just an example). Individuals and demes can also be specialized if needed. Open BEAGLE is a multiple populations EC system where synchronous migration between

[1]The recursive acronym BEAGLE means *the Beagle Engine is an Advanced Genetic Learning Environment*, or in French *Beagle est un Environnement d'Apprentissage Génétique Logiciel Évolué*.

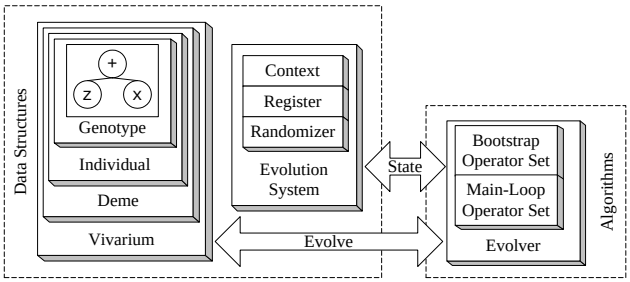

Figure 3. Open BEAGLE Generic EC Framework Architecture.

demes is possible. It takes advantages of greater diversity provided by the simulation of the island-model (see section 4) on one processor.

In the evolution system, the *context* contains the state of the genetic engine, such as the current deme and generation number. This concept is similar to the execution context of a computer. The *register* is a central repository for all evolution parameters. The evolving process itself is governed by an *evolver* that defines sequences of operations, contained in operator sets, that are iteratively applied to demes. The evolver applies the *bootstrap operator set* to initialize the first generation, and the *main-loop operator set* to the subsequent generations. For common EA, standard operators have been defined. These can be common genetic operators, such as selection, crossovers, and mutations, or more functional ones such as statistics calculation, evolution checkpoint backup, and migration operators. This approach is abstract from the evolutionary algorithm used. New algorithms can be made by plugging together standard and custom operators.

The specialized frameworks are at the top level of the architecture. Currently, only classical genetic algorithms and genetic programming frameworks have been implemented. The framework code and documentation is available on the project's Web page at http://www.gel.ulaval.ca/~beagle.

4 Parallel And Distributed Evolutionary Computations

EC is a very generic approach to solve problems. But, applying of EC to real-life problems is usually computationally burdensome, as the CPU time needed to evaluate the fitness of an individual can easily be in the order of seconds or even minutes. However, it is quite easy to exploit the implicit parallelism of EC by distributing the evolutions on several processors.

There are usually four types of Parallel and Distributed Evolutionary Computations (PDEC) [15]: master-slave with one population, island-model made of several distinct populations, fine-grained, and hierarchical hybrids.

Master-slave

Usually, this kind of PDEC has one processor which store the whole population and apply genetic operators (selection, crossover, and mutation). At each generation, the master processor distributes the individuals to slave processors for fitness evaluation. For most difficult problems, fitness evaluation is the main computation bottleneck of EA. This type of PDEC works on a global population and is a simple transposition of the evolution process on several processors.

Island-model

An island-model PDEC consists in evolving isolated sub-populations, named demes, that occasionally exchange individuals in a migration process. These algorithms are very popular in the EC community. Super-linear speedup[2] have been observed when using the island-model [16]. According to Cantú-Paz [15], however, these observations are probably the result of an additional selection pressure induced by the migration process: when the k best individuals of a given deme replace the k worst individuals of another deme, for example. Anyway, using multiple populations with migration schemes usually encourages diversity and prevents premature convergence.

Fine-grained

A fine-grained PDEC consists in evolving populations spatially distributed on processors, generally using a rectangular matrix. Ideally, one individual is assigned to each processor, which allow evaluation of all individuals in parallel. This class of PDEC is particularly adapted to massively parallel SIMD computers. The fine-grained approach is now rarely used in the EC community, as SIMD supercomputers have a low CPU power over cost ratio compared to modern Beowulf clusters [17].

Hierarchical hybrids

This class of PDEC uses an hybrid approach between master-slave and island-model. In hierarchical hybrids, sub-populations evolve on local networks, using the master-slave (or fine-grained) distribution evolution model. At a higher level, the local network populations represent demes of a global island-model. Individuals are exchanged between these demes using migration. Hierarchical hybrids tends to exploit the positive aspects of both master-slave (or fine-grained) and multiple population models.

We found three freely available PDEC systems on the Web: DREAM, ECJ, and GALOPPS. DREAM[3] (Distributed Resource Evolutionary Algorithm Machine) [18] is a peer-to-peer system based on the island-model. In DREAM, each node evolves its own population. Nodes discover the network by interacting with their neighbors. The DREAM system is targeted toward Wide Area Networks (WAN) where communication costs are high [19]. It is very scalable and robust as there is no critical entity that the system depends upon. But it has the limitations of the

[2] Super-linear speedup is observed when the processing time needed for an evolution to converge on n processors is less than $\frac{1}{n}$ the processing time needed on a single processor.

[3] http://www.world-wide-dream.org

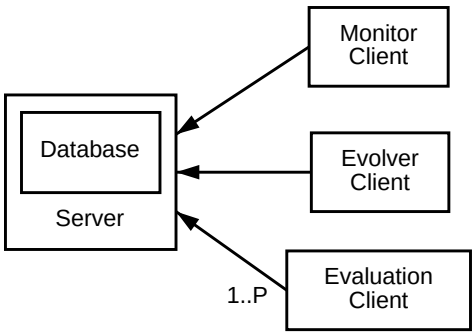

Figure 4. Distributed BEAGLE Architecture.

island-model. ECJ[4] is also based on the island-model. It is a generic EC Java-based framework that implements its PDEC using Java TCP/IP sockets. Its distribution features are not as sophisticated as DREAM, but it includes enough functionalities to be used on Local Area Networks (LAN) or on Beowulf clusters. Finally, GALOPPS[5] (Genetic ALgorithm Optimized for Portability and Parallelism System) is also based on the island-model. It is tightly linked with a specific genetic algorithms library, *S-GA*. It uses the file system to share information among processors. We are not aware of any freely available master-slave PDEC. But we know that several researchers have implemented basic master-slave architectures based on tools such as PVM or MPI.

5 Distributed Beagle

Distributed BEAGLE is an extension of the Open BEAGLE framework which allow evolution distribution on several processors. Evolutions are distributed using a master-slave model. The architecture is independent of the evolutionary algorithm used in Open BEAGLE. Distributed BEAGLE is targeted toward Beowulf clusters or LANs of workstations, where the cost of communication is relatively low.

Distributed BEAGLE comprises four main components: the database, the server, one or more evolver clients, and a pool of evaluation clients. Figure 4 illustrates the system architecture. The system works on data by separating the EC generation concept into two distinct steps: deme evolution and fitness evaluation. Deme evolution is done by evolver clients. It consists in applying several genetic and natural selection operations to evolve the deme through one generation. Once a deme has evolved, the composing individuals need to be evaluated for fitness. Fitness evaluation is done by evaluation clients. When all individuals have been evaluated, the generation is finished and the demes are ready to be evolved again. Since the computational bottleneck in EC is usually the fitness evaluation (at least for hard problems), an evolution with Distributed BEAGLE is usually conducted using a single evolution client and as much evaluation clients as possible (one per available processor).

Database

The database guarantees data persistency by storing the demes and the evolution state. This is an important element of robustness for such a software system, where computations may span several weeks or even months. Furthermore, the use of a common database separates software elements specific to EC from population storage management. Data are classified into two categories: demes that require evolution, and individuals that need evaluation. The database in Distributed BEAGLE is inspired from the distributed and persistent evolutionary algorithm design pattern [20].

Server

The server acts as an interface between the different clients and the database. The primary function of the server is to dispatch the demes to evolver clients, and the individuals to evaluation clients. The number of individuals sent to an evaluation client depends on a load balancing mechanism. This mechanism dynamically and independently adjusts the number of individuals sent to a given evaluation node based on its recent performance history.

Evolver client

An evolver client sends requests for a deme to the server, and then applies selection and genetic operations on this deme. These operations are usually specific to the implemented EC flavor.

Evaluation client

An evaluation client sends requests to the server for individuals that need to be evaluated. The number of individuals returned by the server is variable and depends on the (recent) past performance of the client. The evaluation clients are specific to the problem at hand.

Monitor client

A monitor client sends requests to the server in order to retrieve the current state of the evolution, allowing users to monitor it. This client does not modify the database content.

Load balancing

The load balancing policy is to regulate the size of individual sets in order to achieve (approximately) a constant time period between requests for all evaluation clients. For fast clients, more individuals are sent in order to lower communication latency. For slow clients, fewer individuals are sent in order to minimize synchronization overheads at the end of an evaluation cycle. The pursued time period is set during initialization and can be modified during evolution in order to optimize throughput. This design choice of Distributed BEAGLE allows efficient performances on

[4] http://www.cs.umd.edu/projects/plus/ec/ecj
[5] http://garage.cps.msu.edu/software/galopps

loosely-coupled multi-processor systems such as Beowulf clusters or LAN of workstations.

When all individuals of a deme have been distributed and after a time out proportional to the load balancing time period, individuals that have been sent to lagging nodes are automatically re-dispatched to other nodes by the server until it receives a fitness response. If duplicate answers are received, only the first one is kept and all others are discarded. This approach both reduces the synchronization time needed to finish a generation and assures general fault tolerance for the system.

A prototype of Distributed BEAGLE has been developed using a MySQL[6] database, TCP/IP sockets as the communication protocol and XML (eXtensible Markup Language) [21] for data encoding. The use of XML is inspired from lightweight XML-based protocols for distributed applications such as XML-RPC[7] and SOAP[8]. The main advantages of using XML is that messages are strongly structured, they are represented using portable character encoding, and there is a variety of XML parsers available to process messages.

6 Analysis

In classical master-slave PDEC, two main distribution policies can be used. The first, n-slaves-n-sets, separates the demes into n equal sets of individuals and then sends a set to each of the n nodes. It has the advantage of minimizing the number of client-server connections. But non homogeneous clients generate synchronization overheads as the processing of the fitness evaluation will be as fast as the slowest node. Also, if a node is removed or crashed, its associated set of individuals must be re-dispatched to another node, emphasizing the synchronization overhead. Moreover, a slave added during an evaluation cycle will not be used until the next generation.

The second distribution policy is to send the individuals one-by-one to client nodes. Using this approach, the synchronization problems are minimized and an implicit load-balancing is accomplished. Nodes can be added (or removed) dynamically with minimal overhead. But the problem that lurks is communication latency which reduces scalability of the system.

Distributed BEAGLE exploits a half-way policy where a variable-size set of individuals is sent to evaluation nodes. The speedup of such an approach can be modeled using the following equation.

$$speedup = \frac{T_s}{T_p} \tag{1}$$

where T_s is the time needed to evaluate the fitness of the demes on a single processor, and T_p is the time needed to evaluate the fitness of the same deme in parallel, using P processors. We assume here that the time required for selection and genetic operations (processed by the evolver client) is negligible. Time T_s is given by:

$$T_s = NT_f \tag{2}$$

where T_f is the time needed to evaluate the fitness of a single individual, and N is the population size. Assuming a fixed size S for the sets of individuals (i.e. complexity of the fitness evaluation is about constant over all individuals and processing nodes are homogeneous), and that the number of communication cycles C for a generation is:

$$C = \left\lceil \frac{N}{PS} \right\rceil \tag{3}$$

then time T_p can be modeled by:

$$T_p = \underbrace{CST_f}_{\text{computation}} + \underbrace{CPST_c}_{\text{communication}} + \underbrace{CT_l}_{\text{latency}} \tag{4}$$

where T_c is the transmission time needed to send one individual and receive its fitness, and T_l is the average latency of each connection (here we assume that the load on the server and the network is constant). Finally, a last term T_k may be added to equation 4 to represent the time delay associated with node failure:

$$T_p = CST_f + CPST_c + CT_l + T_k \tag{5}$$

with:

$$T_k = \begin{cases} 0 & K = 0 \\ \underbrace{(1 - 0.5^K)ST_f}_{\text{synchronization}} + \underbrace{KST_c}_{\text{comm.}} + \underbrace{T_l}_{\text{latency}} & K \in [1, P] \end{cases} \tag{6}$$

where K is the number of observed failures. The term associated with synchronization in equation 6 is given under the assumptions that each failure follows a Poisson process and occurs on average at half-way time during an evaluation cycle (i.e. at time $ST_f/2$). We neglect here slowdown associated with processors unavailability, which is likely in presence of failure. It would have greatly complicate the equations while giving no pertinent informations as the loss would be the same independently of the parameters used (i.e. the value of S).

Equipped with this model, we can now investigate some scenarios. Supposing a Beowulf cluster made of identical computers and a 100 Mbit/s Ethernet switch. We evolve populations of $N = 500000$ individuals (total) where fitness evaluation requires an average of $T_f = 1$ s. Assume that the average length of individuals (coded in XML), including their fitness, is 1 Kbyte, which may necessitate about $T_c = 1.4 \times 10^{-4}$ s to transmit over the network[9]. Finally, let the average latency per connection be $T_l = 0.1$ s (this value seems quite high but includes all connection latency, that is latency associated to networking, operating system, and program processing). Figure 5 illustrates the

[6] http://www.mysql.com
[7] http://www.xmlrpc.org
[8] http://www.w3.org/TR/SOAP

[9] This value stems from an effective network bandwidth of ≈ 7 MB/s.

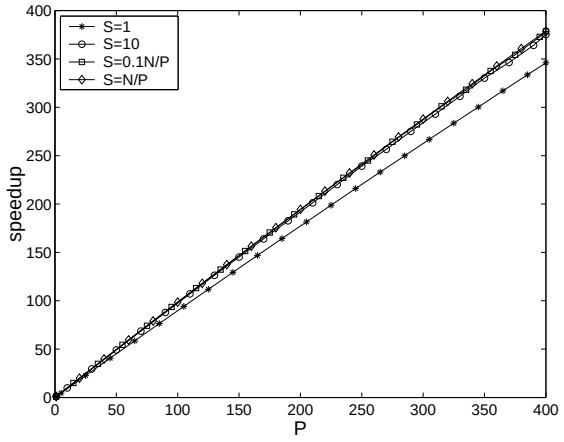

Figure 5. Speedup curves for different set sizes when no failure occur ($K = 0$). Parameters used are $N = 500000$, $T_f = 1$, $T_c = 1.4 \times 10^{-4}$, and $T_l = 0.1$.

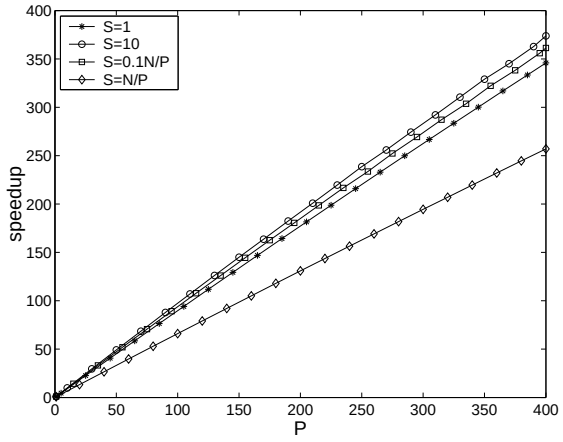

Figure 6. Speedup curves for different set sizes when exactly one failure occurs at each generation ($K = 1$). Parameters used are $N = 500000$, $T_f = 1$, $T_c = 1.4 \times 10^{-4}$, and $T_l = 0.1$.

corresponding speedup curves for different S values and without any failure. This figure shows speedup near linear (near optimal) for $S = \{10, \frac{0.1N}{P}, \frac{N}{P}\}$. For $S = 1$, however, performance degrades significantly given the large latency delay.

Now, Figures 6 and 7 show the same curves but for one and five failures respectively ($K = 1$ and $K = 5$). These Figures show that a value $S = \frac{N}{P}$ no longer achieves linear speedup, and that the intermediary value of $S = 10$ (in this scenario) makes a good compromise between efficiency and robustness. When the number of failure increases, greater loss of performance should be expected with larger sets of individuals. In any case, the second common policy of splitting the total number of individuals into equal size sets ($S = \frac{N}{P}$) is not robust to node failures nor is the first policy efficient for lagging networks. These curves illustrate that the S parameter should be adjusted dynamically in order to optimize performance. In future release this process could even be automated using an intelligent monitor client.

In the above analysis, it was implicitly assumed that the server would never be overwhelmed by the client requests. But in fact, passed a certain threshold on the number of processors, the speedup will reach an asymptote and stagnate with no further performance improvements. This implies that a connection communication time must be less than the time needed to evaluate the fitness of a single set of individuals plus the latency of a connection.

$$SPT_c \leq ST_f + T_l \tag{7}$$

and thus:

$$P \leq \frac{ST_f + T_l}{ST_c} \tag{8}$$

Now, if we set $S = \frac{FN}{P}$, that is a ratio F of the maximum size of individuals set, then the maximum number of

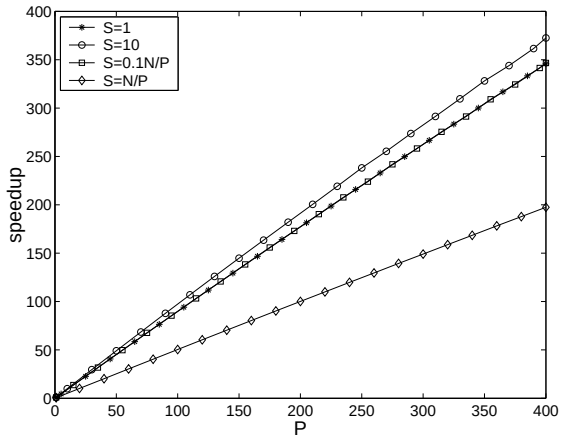

Figure 7. Speedup curves for different set sizes when exactly five failures occur at each generation ($K = 5$). Parameters used are $N = 500000$, $T_f = 1$, $T_c = 1.4 \times 10^{-4}$, and $T_l = 0.1$.

processors becomes:

$$P \leq \frac{FNT_f}{FNT_c - T_l} \tag{9}$$

With the parameters used in our scenario (i.e. $N = 500000$, $T_f = 1$, $T_c = 1.4 \times 10^{-4}$, and $T_l = 0.1$), we obtain a maximum of 7800 processors for $S = 1$, and around 7200 processors for $S = \{10, \frac{0.1N}{P}, \frac{N}{P}\}$. It goes to show that the system can scale well for a relatively high number of processors.

7 Discussion

The island-model is the PDEC that currently receives the most attention in the EC community. Here is a list of the advantages that make this distribution model attractive:

- Scales well as each node communicates only infrequently with its neighbors.
- Robust as there is no centralized control. Data are distributed among nodes; a node crash doesn't affect much the other nodes.
- Communication is asynchronous and limited to punctual migration of small set of individuals.
- Implicit uses of multidemic populations.

But the island-model also has several limitations:

- Population sizes must be tuned to roughly balance computational load of nodes.
- Evolution cannot be reproduced, as migration is asynchronous and depends on the state of the processors/network.
- Distribution of results among nodes complicates data collection and analysis.
- Not particularly adapted to networks of non homogeneous computers where availability of nodes is limited in time.
- When a node crashes, part of the global population doesn't evolve and may even be lost.

On the other hand, master-slave PDEC have also been widely used by the EC community for the following reasons:

- Simple transposition of the single processor evolutionary algorithm onto multiple processor architectures. Allows reproducibility of results.
- No information lost when a slave crashes or is unreachable by the master.
- Appropriate for networks of computers where availability is sometimes limited (i.e. available only during night time). A node can be added or removed dynamically with no loss of information.
- A centralized repository of the population simplifies data collection and analysis.

But the master-slave also has limitations that restrict their usability under some circumstances:

- May not scale as well when the master is overloaded or when the population size becomes very high.
- A crash of the master node can paralyze the whole evolution.
- Significant communication cost as all individuals are transmitted through the network.
- Synchronization overhead for lagging slave nodes[10].

[10] Assuming a generational evolutionary algorithm.

The critical failure point of Distributed BEAGLE is the server. If it crashes, the whole system comes down. But, data persistency is guaranteed by the database. Another interesting aspect of Distributed BEAGLE is the robustness over computer or network failures. If a client (slave) crashes or is unreachable over the network, the data under process is not lost.

An important design choice of Open BEAGLE is the use of a multidemic population model with synchronous migration. This is independent of Distributed BEAGLE. It separates parallel multiple populations from evolution distributed on multiple processors. As far as we know, there is no other such system currently available. We strongly believe that there is no need to evolve a separate deme on every available processor. Moreover, the size of the demes do not needs to be proportional to processor performance. Adding a degree of liberty by separating these elements enable a finer control over the evolution parameters.

8 Conclusion

This paper has presented Distributed BEAGLE, a distribution extension to the Open BEAGLE EC framework. Basically, it is a master-slave PDEC system that implements several features which enhance its general robustness and efficiency:

- Persistent database;
- Dynamically adjustable sets of individuals sent to clients;
- Redistribution of data when clients are lagging or not responding;
- Multidemic populations implemented in a processor independent way.

The Distributed BEAGLE project is still under development. We plan to make it an open source project as soon as the software will be in a stable state, with adequate testing for a beta release. It is targeted for Beowulf clusters and LAN of workstations (also for high performance WAN). It is therefore planned to transform this master-slave architecture into an hierarchical hybrid system, by adding a migration client that will exchange individuals between servers. This will transform master-slave sets into meta-island of a broader evolution environment. Finally, although the current implementation is specialized to do EC, it could possibly be transformed into a more generic framework for distributed computations.

Acknowledgments

The authors would like to thank Frédéric Jean, Jacques Labrie and Hélène Torresan for their participation in the project. This research was supported by a NSERC-Canada and FCAR-Québec scholarships to C. Gagné and a NSERC-Canada grant to M. Parizeau.

References

1. T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, Bristol, UK, 2000.
2. J. Beaulieu, C. Gagné, and M. Parizeau. Lens system design and re-engineering with evolutionary algorithms. In *Proceeding of GECCO 2002*, pages 155–162, New York, NY, USA, 2002.
3. J. R. Koza, M. A. Keane, J. Yu, F. H. Bennett III, and W. Mydlowec. Automatic creation of human-competitive programs and controllers by means of genetic programming. *Genetic Programming and Evolvable Machines*, pages 121–164, 2000.
4. C. Gagné and M. Parizeau. Open BEAGLE: A new versatile C++ framework for evolutionary computation. In *Proceeding of GECCO 2002, Late-Breaking Papers*, New York, NY, USA, 2002.
5. D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, USA, 1989.
6. J. M. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.
7. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
8. W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, 1998.
9. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994.
10. A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.
11. I. Rechenberg. *Evolutionsstrategie*. Friedrich Frommann Verlag (Günther Holzboog KG), Stuttgart, 1973.
12. L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1994.
14. T. Lenaerts and B. Manderick. Building a genetic programming framework: The added-value of design patterns. In *Proceedings of EuroGP'98*, pages 196–208, 1998.
15. E. Cantú-Paz. *Efficient and accurate parallel genetic algorithms*. Kluwer Academic Publishers, Boston, MA, USA, 2000.
16. D. Andre and J. R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.
17. F. H. Bennett III, J. R. Koza, J. Shipman, and O. Stiffelman. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In *Proceedings of the GECCO 1999*, volume 2, pages 1484–1490, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
18. M. G. Arenas, P. Collet, A. E. Eiben, Márk Jelasity, J. J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer. A framework for distributed evolutionary algorithms. In *Proceedings of PPSN 2002*, 2002.
19. M. Jelasity, M. Preuß, and B. Paechter. A scalable and robust framework for distributed applications. In *Proceedings of the CEC 2002*, pages 1540–1545. IEEE Press, 2002.
20. A. Bollini and M. Piastra. Distributed and persistent evolutionary algorithms: a design pattern. In *Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 173–183, Goteborg, Sweden, 1999. Springer-Verlag.
21. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 - W3C recommendation 10-february-1998. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.