A Robust Master-Slave Distribution Architecture for Evolutionary Computations

Christian Gagné, Marc Parizeau, Marc Dubreuil
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

Abstract

This paper presents a new robust master-slave distribution architecture for multiple populations Evolutionary Computations (EC). It discusses the main advantages and drawbacks of master-slave models over island models for parallel and distributed EC. It also formulates a mathematical model of the master-slave distribution policies in order to show that, contrary to what is implied by current mainstream developments in island models, a well designed master-slave approach can be both robust and scalable (up to a certain point). Finally, it introduces some of the details of a new C++ framework named Distributed BEAGLE, which implements this architecture over the Open BEAGLE EC framework.

1 Introduction

The generic problem solving abilities of Evolutionary Computations (EC) are now well established (Bäck et al., 2000). These abilities, however, come at a high computational cost, especially when evaluation of fitness requires a long processing time, which is usually the case for non trivial problems. On the other hand, the evolution process of ECs is implicitly parallel as every individual in the population can be evaluated independently. This paper introduces the architecture of Distributed BEAGLE, a robust, efficient, adaptive, and scalable master-slave model for Parallel and Distributed Evolutionary Computations (PDEC). This architecture is targeted toward local networks of computers. Moreover, it is based on the C++ Open BEAGLE EC framework (Gagné and Parizeau, 2002) which implements multiple populations EC with migration on a single processor.

The recent availability of cheap Beowulf clusters has generated much interest for PDEC. Another often neglected source of CPU power for PDEC are networks of PCs, in many case very powerful workstations, that run idle each day for long periods of time. To exploit efficiently both Beowulfs and networks of heterogeneous workstations, special care must be taken as to not fall into the current patterns of PDEC, as they are often made with Wide Area Network (WAN) in mind. The key features of a good PDEC capable of exploiting networks of heterogeneous workstations are *transparency* for the user, *robustness*, and *adaptivity*. Transparency is essential to both the user of the PDEC and the user of the workstation, as none want to deal with the other. One way to implement such a PDEC is as a screen-saver. Robustness is very important because evolutionary computations may execute over long periods of time during which different types of failures are expected: *hard failures* caused by network problems, system crashes or reboots, and *soft failures* that stem from the use of the workstation for other tasks (e.g. when the user deactivates the screen-saver). Finally, adaptivity refers to the capability of the PDEC to exploit new or compensate for lost computing resources (dynamical network configuration).

Four main types of PDEC can be defined: master-slave with one population, island model made of several distinct populations, fine grained, and hierarchical hybrids (Cantú-Paz, 2000). *Master-slave* PDEC use one processor to store the whole population and apply evolutionary operators (usually selection, crossover, and mutation). The population is distributed to slave processors for fitness evaluation at each generation. *Island model* PDEC consists in evolving isolated demes that occasionally exchange individuals in a migration process. *Fine grained* PDEC consists in evolving populations spatially distributed on processors, generally using a rectangular matrix. This class of PDEC is particularly adapted to massively parallel SIMD computers and is now rarely used in the EC community. Finally, *hierarchical hybrids* use an hybrid approach between master-slave (or fine grained) and island model, in order to exploit positive aspects from both paradigms.

Three freely available PDEC systems can be found on the Web: DREAM, ECJ, and GALOPPS. DREAM (Distributed Resource Evolutionary Algorithm Machine) (Arenas et al., 2002; Paechter et al., 2000) 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 (Jelasity et al., 2002). It is very scalable and robust as there is no critical entity that the system depends upon. But it has the limitations of the island model (see Section 2). ECJ (Luke, 2002) 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 (Genetic ALgorithm Optimized for Portability and Parallelism System) (Punch and Goodman, 2002) 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.

Paper structure goes as follows. First an analysis of the merits and limitations of the two mostly used types of PDEC is made, that is master-slave and island model. Thereafter, different master-slave distribution policies are presented and analyzed. The Open BEAGLE EC framework is then summarized before finishing with details on the *distributed* BEAGLE approach.

2 Master-slave vs Island Model

The island model is the PDEC architecture that currently receives the most attention in the EC community (Andre and Koza, 1996; Paechter et al., 2000) for the following reasons: 1) it scales well as each node communicates only infrequently with its neighbors, 2) the approach is robust as there is no centralized control or data, 3) the communications are asynchronous and limited to punctual migration of small set of individuals, and 4) there is an implicit use of populations with multiple demes.

But the island model also has several limitations: 1) the population sizes must be tuned to roughly balance computational load of nodes, 2) the evolutions cannot be reproduced as migration is asynchronous and depends on the state of the processors/network, 3) the distribution of results among nodes complicates data collection and analysis, 4) the method is not particularly adapted to networks of heterogeneous computers where availability of nodes is limited in time, and 5) when a node crashes, a 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: 1) it is a simple transposition of the single processor evolutionary algorithm onto multiple processor architectures that allows reproducibility of results, 2) there is no permanent loss of information when a slave fails or is unreachable by the master, 3) it is appropriate for networks of computers where availability is sometimes limited (e.g. available only during night time or when screen saver is on) as nodes can be added or removed dynamically with no loss of information, and 4) it is made of a centralized repository of the population which simplifies data collection and analysis.

But the master-slave also has limitations that restrict their usability under some circumstances: 1) it may not scale as well when the master is overloaded or when the population size becomes very high, 2) a crash of the master node can paralyze the whole evolution, 3) there is significant communication cost associated with transmission of all individuals through the network, and 4) there is synchronization overhead when some slave nodes are lagging[1].

In the light of these arguments, we strongly believe that a master-slave architecture for PDEC is appropriate for small to "medium-large" size networks of computers, which are commonly used by EC researchers (see Section 3). Moreover, in the context of heterogeneous and/or partial availability of resources[2] using a master-slave is more natural and efficient than clas-

[1] Assuming a generational evolutionary algorithm.

[2] For example, when using networks of workstations during night time and week-ends, or in the context of very low priority time-sharing.

sical island model PDEC. The classical island-model is not designed to deal with these features, essentially because populations (demes) are tightly coupled with processing nodes. In contrast, the master-slave model has all required features. One issue that needs to be addressed, however, is its ability to scale with a large number of slave nodes, knowing that there exists a communication bottleneck with the master node.

3 Master-slave Distribution Policies

Two main distribution policies can be used in classical master-slave PDEC. 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 heterogeneous clients generate synchronization overheads as the processing of the fitness evaluation will be as fast as the slowest node. If a node is removed or fails in some way, 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 it can also introduce problems of communication latency which, in turn, can reduce scalability of the system.

What we propose in this paper is that a good master-slave PDEC system must use a hybrid distribution policy where a variable-size set of individuals is sent to evaluation nodes. The following reasoning demonstrates the idea by using a mathematical model of the master-slave PDEC with parametrized size of individual sets.

The speedup of such a system can be computed 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 individuals on a single processor, and T_p is the time needed to evaluate the fitness of the same individuals in parallel, using P processors. We assume here that the time required for selection and genetic operations is negligible in comparison. Time T_s is given by:

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

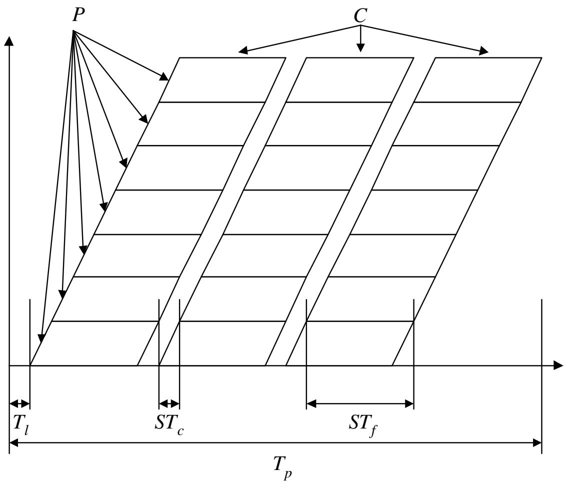


Figure 1: Computation of T_p where C correspond to the communication cycles, P, the number of processors, T_l, the average latency of each connection, T_c, the transmission time needed to send one individual, T_f, the time needed to evaluate the fitness of one individual and S, the number of individuals sent to nodes.

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 sent to processing nodes (i.e. complexity of the fitness evaluation is about constant over all individuals and nodes are homogeneous), then the number of communication cycles C needed for a generation is:

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

Time T_p can now be computed using (worst case):

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

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 network is constant). Figure 1 illustrates the terms of formula 4. Finally, a last term T_k may be added in equation 4 to represent the time delay associated with node failures:

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

with:

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

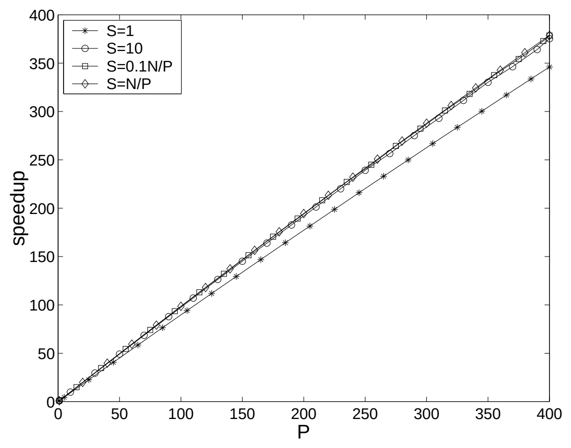
where K is the number of observed failures. The term associated with synchronization in equation 6 is given under the assumptions that each failure is independent and that their occurrences is on average halfway through the fitness evaluation process. We neglect here the fact that node failure might reduce P, the number of available processors. We simply assume that P is constant.

3.1 Results

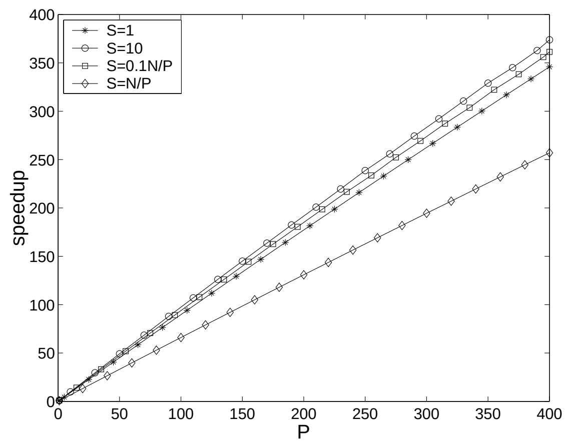
Using this theoretical model, we can now investigate the following very conservative scenario. Given a Beowulf cluster made of identical computers and a 100 Mbit/s Ethernet switch. Say a population of $N = 500000$ individuals (total) is evolved, where fitness evaluation requires an average of $T_f = 1$ s. Let the average length of individuals, including their fitness, be 1 Kbyte, which corresponds to about $T_c = 1.4 \times 10^{-4}$ s if the effective network bandwidth is ≈ 7 MB/s. Finally, let the average latency per connection be $T_l = 0.1$ s (this value may seem quite high but includes all connection latency, that is latency associated with networking, operating system, and program processing).

Figure 2a presents the speedup curves for some S values. It shows that the speedup is near optimal for $S = \{10, \frac{0.1N}{P}, \frac{N}{P}\}$. For $S = 1$, however, performance starts to degrade given the large latency T_l. Figures 2b and 2c show the same curves but for one and five failures respectively ($K = 1$ and $K = 5$). It can be observed that a value $S = \frac{N}{P}$ no longer achieves linear speedup, and that the intermediary value of $S = 10$ (for this scenario) makes a good compromise between efficiency and robustness. When the number of failures increases, greater loss of performance should be expected with larger sets of individuals.

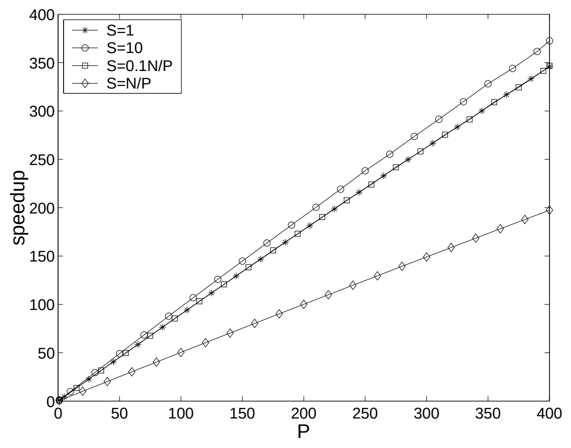
Figure 3 investigate the effect of changing times T_f, T_c, and T_l on the speedup. First, assume a fixed number of processors $P = 200$. Figure 3a presents the effect of varying the fitness evaluation time T_f in the range $[0.1, 10]$. It shows that the shorter is T_f, the bigger is the influence of T_c and T_l, which is even more noticeable when S is very small. Figure 3b shows the effect of communication time on the speedup ($T_c \in [1.4 \times 10^{-5}, 1.4 \times 10^{-3}]$). As expected, the bigger the communication time, the smaller is the speedup. But, we notice that the decrease of speedup is proportionally the same for the different values of S. This make sense as the amount of communication is independent of S. Finally, Figure 3c presents the speedup observed for different latencies ($T_l \in [0.01, 1.0]$). It clearly demonstrates that the influence of latency becomes important as S gets smaller. From this obser-



(a)

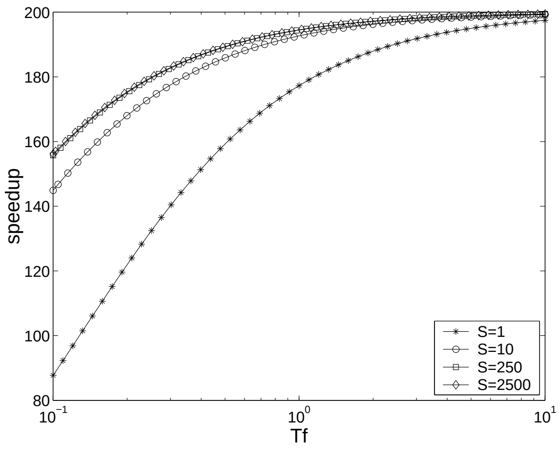


(b)

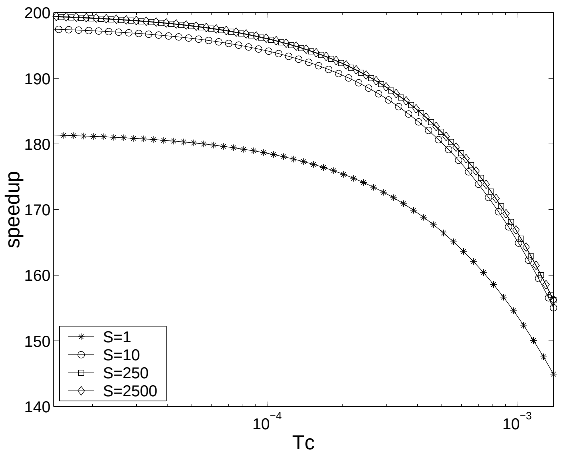


(c)

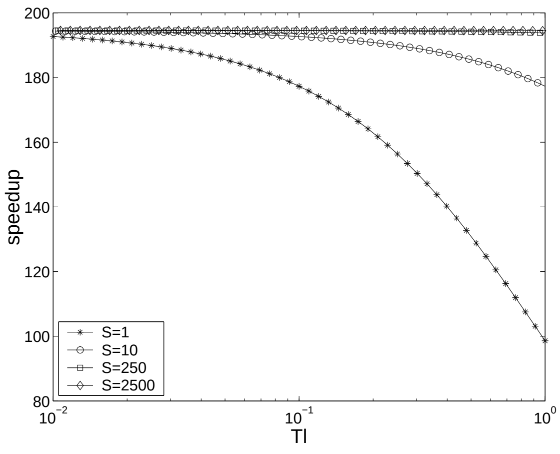
Figure 2: Speedup curves for different size of sets $S = \{1, 10, \frac{0.1N}{P}, \frac{N}{P}\}$, as a function of the number of processor used $P \in [1, 400]$: (a) when no failure occurs ($K = 0$), (b) when exactly one failure occurs ($K = 1$), and (c) when exactly five failures occur ($K = 5$). Other parameters are $N = 500000$, $T_f = 1$ s, $T_c = 1.4 \times 10^{-4}$ s, and $T_l = 0.1$ s.



(a)



(b)



(c)

Figure 3: Speedup curves for different size of sets $S = \{1, 10, \frac{0.1N}{P}, \frac{N}{P}\}$, as a function of time (logarithmic scale): (a) fitness evaluation time $T_f \in [0.1, 10]$ s, (b) communication time $T_c \in [1.4\times10^{-5}, 1.4\times10^{-3}]$ s, and (c) latency time $T_l \in [0.01, 1.0]$ s. Other parameters are $N = 500000$, $P = 200$, $T_f = 1$ s, $T_c = 1.4 \times 10^{-4}$ s, and $T_l = 0.1$ s when they are not the independent variable.

vation, one should conclude that S should be as large as possible in order to minimize performance degradation caused by connection latency in loosely-coupled parallel systems, but no larger.

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 ($S = 1$) efficient for lagging networks. These curves thus globally show that parameter S should be adjusted dynamically in order to optimize performance.

3.2 Scalability Limitations

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 number of processors, the master node networking capacity (bandwidth) will saturate. At this point, the speedup will no longer improve with the addition of new processors. This implies that overall communication time (including latency) must be less than the time needed to evaluate fitness:

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

and thus:

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

Now, if $S = \frac{FN}{P}$, that is a fraction F of the maximum size of individual sets, then the maximum number of 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 an upper limit of around 7200 processors for $S = \{10, \frac{0.1N}{P}, \frac{N}{P}\}$, and 7800 processors[3] for $S = 1$, which translate for all cases to a speedup of about 3500, a little less than half of the optimal value. It goes to show that the system can scale well for a relatively high number of processors. In the light of this result, the master-slave PDEC cannot be overlooked anymore in fear of network bottleneck.

4 Proposed Implementation

Distributed BEAGLE, the proposed system, is a master-slave distribution architecture with variable

[3]This difference is explained by the important latency delays when $S = 1$.

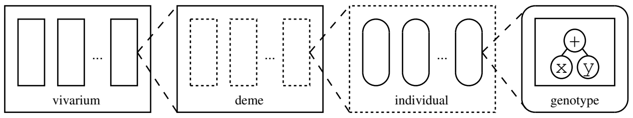


Figure 4: Open BEAGLE Population Structure.

size of distribution sets. It is designed as a distribution extension of the C++ EC framework Open BEAGLE, but is generic enough, and modular, to be used in conjunction with others EC systems with some minor modifications. To better understand the distribution architecture, here is the pertinent background on Open BEAGLE.

4.1 Open BEAGLE

Open BEAGLE[4] (Gagné and Parizeau, 2002) is a C++ framework for doing almost any kind of EC. Its architecture follows the principles of object oriented programming, where some abstractions are represented by loosely coupled objects and where it is common and easy to reuse code. Currently, classical genetic algorithms and genetic programming have been implemented in the framework. The framework code and documentation is available on the project's Web page[5].

The population in Open BEAGLE is organized into a four-level structure as illustrated in Figure 4. The vivarium is a container for demes of individuals. The individuals themselves are containers for an abstract genotype. This genotype can be instantiated to any relevant structure (in Figure 4, it is shown as a genetic programming tree, but this is just an example). Individuals and demes can also be specialized if needed. On the other hand, the evolutionary algorithms are constructed by defining sequences of operations on demes. For example, these operations could simply be selection, crossover, and synchronous migration. The use of a hierarchy of abstract data structures enables the ontogeny of multiple populations EC system where synchronous migration between demes is possible. It is thus generic enough to simulate an island model on a single processor.

4.2 Distributed BEAGLE

Distributed BEAGLE comprises four main components: the database, the server, one or more evolver clients, and a pool of evaluation clients. Figure 5 illustrates the system's architecture. The system works on

[4]The recursive acronym BEAGLE means *the Beagle Engine is an Advanced Genetic Learning Environment*.

[5]http://www.gel.ulaval.ca/~beagle

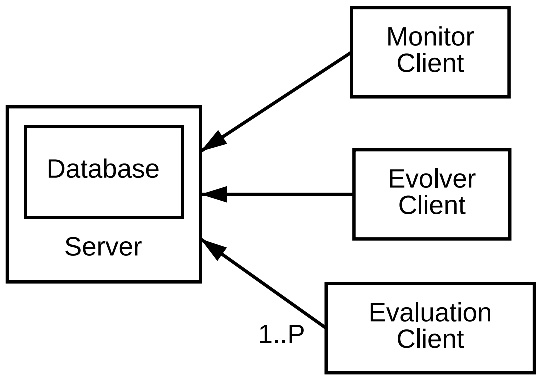


Figure 5: Distributed BEAGLE Architecture.

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).

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 use of a database in Distributed BEAGLE is inspired from the distributed and persistent evolutionary algorithm design pattern (Bollini and Piastra, 1999).

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. The mechanism dynamically and independently adjusts the number of individuals sent to a given evaluation node based on its recent performance history.

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.

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.

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.

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 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.

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) fails or is unreachable over the network, the data it was processing is not permanently lost.

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

A prototype of Distributed BEAGLE has been developed using a MySQL database[6], TCP/IP sockets as the communication protocol and XML (eXtensible Markup Language) (Bray et al., 1998) 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 them. The 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.

5 Conclusion

This paper presents the benefit of using master-slave architecture for PDEC on LANs of workstations and Beowulf clusters. With a mathematical model and plausible scenarios, it demonstrates that a master-slave architecture with variable size of distribution sets constitutes an excellent trade-off between efficiency and robustness. It is therefore argued that the use of master-slave with such distribution policy is appropriate for networks of heterogeneous computers where availability is limited and/or variable, as when using computers during night time or in a time-sharing way with low priority processes, or even as a screen saver in LANs of computers, when their availability is unknown beforehand.

Promoting a master-slave approach may seems a little odd as most PDEC researchers are pushing toward the use of island model distribution architecture. The island model is a powerful PDEC. Its main advantages is to minimize communication costs while allowing implicit use of multiple deme populations. But it has the drawback of using at least one population per processor. Population size must therefore be tuned beforehand for processor capacity, which is not particularly adapted on networks with limited availability. Moreover, if the problems tackled with EC require very complex (long) fitness evaluations, the size of each island may need to be reduced, which is bad for diversity. The island model PDEC was made with long period of computers availability in mind. If a popu-

[6]http://www.mysql.com

[7]http://www.xmlrpc.org

[8]http://www.w3.org/TR/SOAP

lation cannot evolve because it can't have CPU time, a more permissive PDEC should be used. There is no need to stick to an island model architecture only to benefit from greater diversity provided by multiple populations evolving in parallel with migration. We believe that it is highly desirable to separate the *parallel* from the *distributed* in PDEC. Furthermore, it is always possible to transform a master-slave architecture into a hierarchical hybrid, by allowing migration between populations. This will transform master-slave sets into meta-island of a broader evolution environment, allowing astonishing scalability.

This paper also introduced Distributed BEAGLE, a distribution extension to the Open BEAGLE EC framework. Basically, it is a master-slave PDEC system with variable-size distribution sets. The system implements several features that 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, and populations with multiple demes implemented in a processor independent way.

Acknowledgments The authors would like to thank Frédéric Jean, Jacques Labrie and Hélène Torresan for their participation in the development of the first prototype of Distributed BEAGLE. This research was supported by an NSERC-Canada and FQRNT-Québec scholarships to C. Gagné and an NSERC-Canada grant to M. Parizeau.

References

Andre, D. and Koza, J. R.: 1996, Parallel genetic programming: A scalable implementation using the transputer network architecture, in P. J. Angeline and K. E. Kinnear, Jr. (eds.), *Advances in Genetic Programming 2*, Chapt. 16, pp 317–338, MIT Press, Cambridge, MA, USA

Arenas, M. G., Collet, P., Eiben, A. E., Jelasity, M., Merelo, J. J., Paechter, B., Preuß, and Schoenauer, M.: 2002, A framework for distributed evolutionary algorithms, in *Proceedings of PPSN 2002*

Bäck, T., Fogel, D. B., and Michalewicz, Z. (eds.): 2000, *Evolutionary Computation 1: Basic Algorithms and Operators*, Institute of Physics Publishing, Bristol, UK

Bollini, A. and Piastra, M.: 1999, Distributed and persistent evolutionary algorithms: a design pattern, in *Proceedings of EuroGP'99*, Vol. 1598 of *LNCS*, pp 173–183, Springer-Verlag, Goteborg, Sweden

Bray, T., Paoli, J., and Sperberg-McQueen, C. M.: 1998, *Extensible Markup Language (XML) 1.0 - W3C Recommendation 10-February-1998*, Technical Report REC-xml-19980210, World Wide Web Consortium

Cantú-Paz, E.: 2000, *Efficient and accurate parallel genetic algorithms*, Kluwer Academic Publishers, Boston, MA, USA

Gagné, C. and Parizeau, M.: 2002, Open BEAGLE: A new versatile C++ framework for evolutionary computation, in *Proceeding of GECCO 2002, Late-Breaking Papers*, New York, NY, USA

Jelasity, M., Preuß, M., and Paechter, B.: 2002, A scalable and robust framework for distributed applications, in *Proceedings of the CEC 2002*, pp 1540–1545, IEEE Press

Luke, S.: 2002, *ECJ Evolutionary Computation System*, Seen January 07, 2003 at http://www.cs.umd.edu/projects/plus/ec/ecj

Paechter, B., Bäck, T., Schoenauer, M., Sebag, M., Eiben, A. E., Merelo, J. J., and Fogarty, T. C.: 2000, A distributed resource evolutionary algorithm machine (DREAM), in *Proceedings of CEC'00*, pp 951–958, IEEE Press

Punch, B. and Goodman, E.: 2002, *GALOPPS 3.2*, Seen December 10, 2002 at http://garage.cps.msu.edu/software/galopps