Analysis of a Master-Slave Architecture for Distributed Evolutionary Computations

Marc Dubreuil, Christian Gagné, and Marc Parizeau

***Abstract*— This paper introduces a new mathematical model of the master-slave architecture for distributed Evolutionary Computations (EC). This model is validated using a concrete implementation based on the Distributed BEAGLE C++ framework. Results show that contrary to (current) popular belief, master-slave architectures are able to scale well over LANs of workstations using off-the-shelf networking equipment. The main properties of the master-slave are also compared with those of the more mainstream island-model.**

***Index Terms*— Master-Slave Architecture, Evolutionary Computations, Distributed BEAGLE.**

I. Introduction

The generic problem-solving abilities of Evolutionary Computations (EC) are now well established [1], [2], [3]. These abilities, however, come at a high computational cost. Under the assumption that fitness evaluation time is high for "real-world" EC problems, the aim of this paper is to show that a master-slave architecture can be designed to efficiently distribute the computation load over a Local Area Network (LAN) of dedicated workstations (the so-called Beowulf cluster). It will be shown that this architecture can be made both robust and efficient, as well as adaptive to provide load balancing in a dynamic environment. Moreover, through a realistic mathematical model, it will also be shown experimentally and theoretically that the proposed architecture can scale well over a large number of processors using off-the-shelf networking equipment. Called Distributed BEAGLE, the developed distributed EC system is integrated transparently with the C++ Open BEAGLE framework [4].

The recent availability of cheap Beowulf clusters has generated much interest for Parallel and Distributed Evolutionary Computations (PDEC) [5], [6]. 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*. The basic master-slave PDEC uses one processor (the master) to manage the population and apply evolutionary operators (selection, crossover, and mutation), and a set of slave processors to distribute the fitness evaluation task. An island-model PDEC consists in evolving isolated sub-populations (sometimes called "demes"), one on each processor, while occasionally exchanging individuals between islands in a migration process. The so-called 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 (Single Instruction Multiple Data) computers and is now rarely used in the EC community. Finally, hierarchical hybrids use a hybrid approach between master-slave (or fine-grained) and island-model, in order to exploit positive aspects from both paradigms. In this paper, we concentrate on the master-slave paradigm, even though we do compare the properties of the master-slave and island-model in the next section, in order to emphasize their respective merits.

In the last few years, several freely available PDEC systems have emerged on the Internet, such as DREAM, ECJ, ParadisEO, JDEAL, GALOPPS, or Paladin-DEC (some of them may not be actively supported or updated). DREAM (Distributed Resource Evolutionary Algorithm Machine) [7], [8], [9] is a peer-to-peer system based on the island-model. In DREAM, each node evolves its own population. Nodes can discover the complete network through a "push-pull anti-entropy epidemic" algorithm. The DREAM system is targeted toward Wide Area Networks (WAN) where communication costs are high. It assumes an application which is massively parallelizable, asynchronous and robust (i.e. its success does not depend on the success of any sub-process), that requires little communication between sub-processes, and has large resource requirements. ECJ [10] is a generic EC Java-based framework that includes some PDEC components using Java TCP/IP sockets. Distribution components of ECJ allow both the island-model and the master-slave strategies. ECJ's distribution features are not as sophisticated as DREAM, but they are sufficient for exploiting the computer resources of a Local Area Network (LAN). ParadisEO (PARAllel and DIStributed Evolving Objects) [11] can also be used for both island-model and master-slave configurations. Although it is designed as an extension of the Evolving Object (EO) framework, it is not limited to EC as it can also perform local search algorithms. The framework is coded in C++ using MPI/PVM for message passing and the pthread library for multi-threading. JDEAL (Java Distributed Evolutionary Algorithms Library) [12] is a master-slave architecture coded in Java. It integrates its own implementation of genetic algorithms and evolution strategies. GALOPPS (Genetic ALgorithm Optimized for Portability and Parallelism System) [13] is a C implementation of the island-model, tightly linked with the $S - GA$ genetic algorithms library. Paladin-DEC [14] is another Java implementation of the island-model. It integrates its own version of genetic algorithms, genetic programming, and evolution strategies, with dynamic load balancing and fault tolerance.

Although most of these systems seem fine on paper, it is interesting to note that their authors have reported speedup

This work was supported by NSERC-Canada and FQRNT-Québec. The authors are from the Laboratoire de Vision et Systèmes Numériques (LVSN), Département de Génie Électrique et de Génie Informatique, Université Laval, Québec (QC), Canada, G1K 7P4. Contact: parizeau@gel.ulaval.ca

performances only for modest numbers of processors. Indeed, most results are given for fewer than 10 processors. For instance, in the island-model category, the authors of DREAM report a speedup of 86% for 9 processors. They also report some results for experiments with 20 processors where some simulate crashes. Even though the island-model PDEC should in theory be able to scale up very well, it is tempting to conclude from this lack of large scale results that it is not so easy to exploit in practice, essentially because hard EC problems (those that require distributed solutions) tend to need small numbers of large sub-populations more often than large numbers of small ones [15]. In the master-slave category, the authors of Paladin-DEC report a speedup of 67% for 5 processors while the authors of ParadisEO report a speedup of 83% for 40 processors (this is the exception to the fewer than 10 processors rule). At this point, even though an intrinsic architectural bottleneck exists when the master becomes overwhelmed by slave requests and communication overheads, we argue that the master-slave architecture is capable of scaling up well with hundreds of processors (using yesterday's computers and networks), if not for thousands of processors (using today's technology), depending on how hard the fitness is to evaluate relative to the communication overheads, for real-world applications.

We should mention also that in the late 1990's and early 2000's, two Beowulf clusters of respectively 70 and 1000 nodes were built by Koza especially for PDEC [16], [17], using off-the-shelf components. These clusters were used to publish many research papers, but the software used to run them was never made public nor described in detail. It is however of the island-model type. Typically, the later cluster was used to evolve 1000 demes of $10\,000$ genetic programs in parallel (genetic programming is known to perform best with very large populations). It appears that speedups of about 100% were achieved in this context (because communications between adjacent cluster nodes are sparse), although we could not find explicit numbers in Koza's vast body of literature.

An initial version of the work presented hereafter was first published as a late breaking paper at the GECCO 2003 conference [18], but without any experimental results. Since then, work was completed on our software implementation, the mathematical model of the master-slave architecture was completely revised, and thorough experiments were conducted. The rest of the paper is structured as follows. In Section II, the properties of the master-slave and island-model are compared more precisely in order to put their respective merits into perspective. Then, Section III presents the mathematical model of the master-slave proposed for speedup evaluation. This model can be used to predict the speedup of any configuration given the number of processor slaves, the (effective) network throughput and latency, the average time for evaluating the fitness of an individual, and the average time for marshalling an individual and its fitness value. Results are reported in Section IV, both theoretical and experimental. They show that the proposed model is realistic enough to enable extrapolation. Finally, a brief overview of the system implementation is given in Section V, before drawing conclusions in Section VI.

II. Master-slave vs island-model

The island-model is the PDEC architecture that currently receives the most attention in the EC community [8], [19] for the following reasons: 1) it scales up 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 sets of individuals; and 4) there is an implicit use of populations with multiple demes.

But the island-model also has several limitations: 1) population sizes must be tuned to balance computational load of nodes; 2) evolutions cannot be reproduced as migration is asynchronous and depends on the state of the processors/network; 3) 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 does not evolve and may even be lost.

On the other hand, the master-slave has also been 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 a 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 up as well when the master is overloaded or when 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, assuming a generational evolutionary algorithm.

As mentioned previously, most papers that measure speedup improvement almost always use fewer than ten processors (or ten subpopulations) as in [14], [15], and [20]. A minority of researchers need more subpopulations. Few ever use more than 25 processors [11]. Considering the time needed to process large populations or complex fitness evaluation functions, increasing the number of processors is vital. Proving that one architecture scales up well to 10 processors does not prove that it can scale up for 100 processors or more. Also, if there is a possibility that the island-model can produce super-linear speedups induced by the migration process [5], it should be noted that this feature can be easily simulated in a master-slave environment.

In light of these arguments, we strongly believe that a master-slave architecture for PDEC is appropriate for Beowulf clusters or LANs of workstations which are commonly available to EC researchers within their institutions. Moreover,

in the context of heterogeneous and/or partial availability of resources, for example when using networks of workstations during night time and week-ends, or in the context of very low priority time-sharing, using a master-slave is more natural and efficient than a classical 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 of the required features. One issue that needs to be addressed, however, is its ability to scale up with a large number of slave nodes, knowing that there exists a communication bottleneck with the master node.

III. Master-slave model

Recent literature on mathematical models for master-slave PDECs is somewhat sparse. The most important is the book from Cantú-Paz [5] which covers many aspects of PDECs. In particular for the master-slave, it defines a simple model based on the p-slaves-p-sets policy, where the individuals of a given generation are separated into p sets of equal size, and dispatched to p slaves for fitness evaluation.

The speedup of such a system can be computed using the following simple formula.

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

where T_s is the time needed to evolve a population of individuals on a single processor (serial computer), and T_p is the time needed to evolve the same population in parallel, using p slave processors (parallel computer). Time T_s is given by:

$$T_s = n[t_e + t_f], \tag{2}$$

where n is the population size, t_e is the average time required for applying selection and genetic operations per individual (to evolve one individual), and t_f is the average time needed to evaluate its fitness.

According to Cantú-Paz, time T_p can be approximated by:

$$T_p = n\,t_e + pt_c + \frac{nt_f}{p}, \tag{3}$$

where t_c is the communication time needed for transmitting a set of individuals to one slave. Term pt_c thus represents the time required to send the p sets to the p slaves, while term nt_f/p is the time required by the last slave to evaluate the fitness of its n/p individuals. Using this model he then computes the optimal number of slaves, that is the point at which the increase in communication time surpasses the decrease of fitness evaluation time. But this model neglects the possibility of parallelism between the master and the slaves. Equation (3) assumes that the master cannot do anything else while sending sets of individuals to slaves, and that slaves need to receive their complete set of individuals before starting their fitness evaluation task. In fact, if we consider multiple communication cycles between master and slaves, where smaller sets of individuals are sent for each cycle, then we can show that there exists an optimal number of cycles.

Assuming t_f is large enough, (3) can be re-written as follows with α designating the number of server cycles:

$$\begin{aligned} T_p &= n\,t_e + \frac{(p-1)\,t_c}{\alpha} + \sum_{1}^{\alpha}\left[\frac{t_c}{\alpha} + \frac{n\,t_f}{\alpha p}\right] \\ &= n\,t_e + \frac{(p-1)\,t_c}{\alpha} + t_c + \frac{n\,t_f}{p}. \end{aligned} \tag{4}$$

Equations (3) and (4) are equivalent when $\alpha = 1$. When $\alpha > 1$, (4) specifies that the first cycle is the most costly because some work needs to be assigned to the first $p-1$ slaves before the last slave is able to start its own work. Thereafter (during subsequent cycles), the t_f large assumption implies that the master will be able to provide more work to slaves without delay. Otherwise, either the master or the network would be overwhelmed. Thus, the global fitness evaluation task completes when the last slave completes its part of the workload (i.e. after $t_c + n\,t_f/p$).

By setting $\partial T_p/\partial\alpha = 0$ and solving for α, we obtain $\alpha \to \infty$ which means that individuals should be sent to slaves one by one. Of course, this assumes that there is no latency in communications. By adding a latency term t_l in (4), T_p becomes:

$$T_p = n\,t_e + \frac{(p-1)\,t_c}{\alpha} + (\alpha + p - 1)\,t_l + t_c + \frac{n\,t_f}{p}, \tag{5}$$

where p latencies are introduced by the first cycle (slave requests are synchronous), and $\alpha - 1$ by subsequent cycles (assuming $t_c < t_f$). Equation (5) can be optimized by:

$$\alpha = \sqrt{\frac{(p-1)\,t_c}{t_l}}. \tag{6}$$

This result demonstrates that the p-sets-p-slaves policy developed by Cantú-Paz is not optimal. It shows that the number of communication cycles should be proportional to both the number of slaves (p) and the communication time (t_c), and inversely proportional to latency (t_l).

The model proposed in this paper takes a different approach. First, its formulation is recursive in order to take into account non linear delays. Second, it introduces new time parameters to represent marshalling and unmarshalling of individuals. These variables introduce latencies that are somewhat different in nature than network latencies that are mostly I/O bound. Network latencies allow some parallelism whereas marshalling latencies are CPU bound. Moreover, they may or may not be negligible depending on the type of EC. For example, genetic programming may induce important marshalling latencies. We also distinguish the mean transmission time needed for sending one individual over the network (t_{xi}) and for receiving its fitness evaluation (t_{xf}). It is further assumed that the master-slave architecture follows a server-client paradigm where connections are closed after each request. A slave (client) connects to the master (server) to request work; and the master responds by sending a set of μ individuals that need fitness evaluation. The slave then processes these individuals and reconnects to the master to both return the computed fitness values and request more work.

The marshalling process consists in converting the internal representation of an object into a stream of data for sending

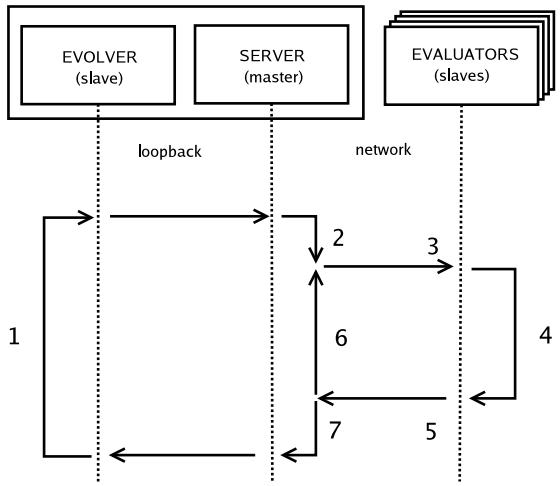

Fig. 1. Sequence diagram for the proposed master-slave model: step 1 requires time $n[t_{ui}+t_{uf}+t_e+t_{mi}]$; step 2 is nt_{ui}; step 3 is $t_l+\mu[t_{mi}+t_{xi}]$; step 4 is $\mu[t_{ui}+t_f]$; step 5 is $t_l+\mu[t_{mf}+t_{xf}]$; step 6 is μt_{uf}; and step 7 is $n[t_{mi}+t_{mf}]$.

them over a network. The unmarshalling process is the reverse operation. The times required to marshal and unmarshal individuals will be denoted by t_{mi} and t_{ui}, while those of the fitness values will be written t_{mf} and t_{uf}.

The sequence diagram of Figure 1 illustrates the proposed model which is constructed around three process types: an *evolver* slave, a master *server*, and a set of *evaluator* slaves. The evolver is responsible for applying the selection and genetic operators while the evaluators are used exclusively for computing fitness. The server is application independent. Its role is to coordinate the work of the evolver and evaluators. The evolver and server processes are assumed to run on the same machine but this is not a limitation.

At the start of a generation, the system is at position 1 (see Figure 1). The evolver unmarshals the individuals (and their fitness values) received from the server, applies selection and genetic operators, and re-marshals the new individuals to send them through the loopback network interface. This requires $n[t_{ui}+t_{uf}+t_e+t_{mi}]$ units of time. The server, now at position 2, receives the individuals and unmarshals them in time nt_{ui}. When an evaluator slave connects at position 3, the server transmits a set of μ individuals in time $t_l+\mu[t_{mi}+t_{xi}]$. The evaluator at position 4 unmarshals the individuals and computes their fitness in time $\mu[t_{ui}+t_f]$. The evaluator then reconnects at position 5, marshals the computed fitness values, and sends them back to the server in time $t_l+\mu[t_{mf}+t_{xf}]$. The server at position 6 now unmarshals the received fitness values in time μt_{uf} and loops to step 3 until every individual has been processed. When complete, the server marshals all individuals and their fitness values, and sends them to the evolver in time $n[t_{mi}+t_{mf}]$ at step 7. The evolver now restarts at position 1.

In the following equations, let μ represent the desired size of sets for evaluator slaves, and $\lceil n/\mu \rceil \geq p$, let $T_1(i)$ denote the time point at which the server completes everything that it needs to do before slave $i \bmod p$ can begin processing its $\lceil i/p \rceil$ cycle ($\lceil x \rceil$ specifies the smallest integer greater or equal to x), and let $T_2(i)$ denote the time point at which slave $i \bmod p$ ends its $\lceil i/p \rceil$ cycle. Then, time T_p can be expressed as follows:

$$T_p = \max_{j=1\ldots p}\left[T_2\left(\left\lceil\frac{n}{\mu}\right\rceil - j + 1\right) + m\left(\left\lceil\frac{n}{\mu}\right\rceil - j + 1\right) t_{uf}\right] + n[t_{mi}+t_{mf}], \quad (7)$$

where:

$$m(i) = \begin{cases} \mu & i \leq \lfloor n/\mu \rfloor \\ n \bmod \mu & \text{otherwise} \end{cases} \quad (8)$$

is equal to μ except for the last slave of the last cycle which receives fewer individuals if n is not a multiple of μ.

The time to complete one generation is the end time of the slowest slave of the last cycle, plus the time to unmarshal the results, and the time needed by the server to return the evaluated population to the evolver (i.e. $n[t_{mi}+t_{mf}]$). Time T_2 itself can be expressed by the following formula:

$$T_2(i) = T_1(i) + m(i)[t_{ui}+t_f+t_{mf}+t_{xf}] + t_l \quad (9)$$

$$T_1(1) = n[t_{ui}+t_{uf}+t_e+t_{mi}+t_{ui}] + t_l + \mu[t_{mi}+t_{xi}] \quad (10)$$

$$T_1(i) = t_l + m(i)[t_{mi}+t_{xi}] + \begin{cases} T_1(i-1) & 1 < i \leq p \\ \max\begin{bmatrix} T_1(i-1), \\ T_2(i-p) \end{bmatrix} + \mu t_{uf} & \text{otherwise} \end{cases} \quad (11)$$

For the special case of the first evaluator slave of the first cycle (Eq. 10), before it can start processing individuals, it must wait for the evolver to complete its work (in time $n[t_{ui}+t_{uf}+t_e+t_{mi}+t_{ui}]$), it must wait for the server to establish a connection (in time t_l), and it must wait for the server to send the individuals themselves after they have been marshalled and transmitted through the network (in time $\mu[t_{mi}+t_{xi}]$). In the general case of (11), only the latter two tasks need to be accomplished. The server must establish a connection (in time t_l) and transmit the individuals in time $m(i)[t_{mi}+t_{xi}]$. To this delay, we need to add one of two possible additional delays. The first delay is during the first cycle. In that case, we simply add the time required by the master to serve the previous slave (i.e. $T_1(i-1)$). The second possible delay is for subsequent cycles. In that case (i.e. $i > p$), the slave must wait for either the start time of the previous slave (i.e. $T_1(i-1)$) or its own end time for the previous cycle (i.e. $T_2(i-p)$), and the server must eventually unmarshal the fitness values returned by the slave for the previous cycle (in time μt_{uf}).

IV. Theoretical and Experimental Results

Using the proposed mathematical model, we can now investigate the following realistic scenario. Consider a Beowulf cluster made of homogeneous computers and a 100 Mbits/sec Ethernet switch. Assume a population of $n = 100\,000$ individuals (total) is evolved, where fitness evaluation requires $t_f = 0.25$ sec/individual on average. Let the average length of (marshalled) individuals be 150 bytes and their fitness require 50 bytes, which corresponds to about $t_{xi} = 4.54 \times 10^{-5}$ sec and $t_{xf} = 1.52 \times 10^{-5}$ sec assuming an effective network bandwidth of ≈ 3.3 MBytes/sec. The times to marshal and

unmarshal individuals are $t_{mi} = 9.5 \times 10^{-6}$ sec and $t_{ui} = 9.6 \times 10^{-5}$ sec while the times to marshal and unmarshal fitness values are $t_{mf} = 6.5 \times 10^{-6}$ sec and $t_{uf} = 6.45 \times 10^{-5}$ sec. Finally, let the average latency per connection be $t_l = 0.001$ sec. Note that these values are realistic since they have been estimated from our experimental setup (described below).

Given these values, Figure 2a presents the theoretical (predicted) speedup curves for three values of μ that correspond respectively to 1, 10, and 100 server cycles per generation ($\mu = \{\lceil n/p \rceil, \lceil n/10p \rceil, \lceil n/100p \rceil\}$). Figure 2b gives the same speedup curves but this time using the Distributed BEAGLE environment (see Section V for more information). These experimental curves are average speedups for three independent simulation runs. They were obtained using a Beowulf cluster of 24 AMD Athlon 1.2GHz nodes running a special GA application where the fitness evaluation function does nothing except to unmarshal the individuals, sleep for $m(i)t_f$ seconds, marshall random fitness values, and return these to the master. This way, we are able to run many evaluator slaves on each cluster node without taxing its CPU too much (for $p = 1000$, we run up to 42 processes on each node; the CPU load never rose above 25%). Because we evaluate the speedup over a single generation, it is independent of whether slaves are actually crunching numbers or simply sleeping, it only depends on the time that they take to return their response. To make a more realistic simulation, we have also tried a random evaluation time using a Gaussian distribution $N(t_f, t_f/3)$, instead of the default $N(t_f, 0)$. Speedup results are mostly unchanged within a $\pm 3\%$ margin.

As can be seen from Figure 2, the predicted speedups fit very nicely with the observed experimental speedups, up to about 500 processors at which point the server becomes overwhelmed. This bottleneck occurs sooner than predicted essentially because the server needs to accomplish certain bookkeeping tasks that are not currently modelled. As for the 100 cycles curve, it is interesting to note that with more than 500 processors, some of them will not receive more than 1 individual per cycle. When more than 1000 processors are used, they will receive either 1 or 0 individual, which explains the flat part of the curve in Figure 2a.

Even though differences are small for the chosen parameter combination, this figure also illustrates that using the $\mu = n/p$ policy (1 cycle) may not be optimal depending on scale, latency, transmission time, and fitness evaluation time. Figure 2c shows that fitness evaluation time can affect performance greatly for this configuration when using 100 processors. At $t_f = 0.01$ sec, speedup is around 14% of the optimal (for 10 cycles), while it reaches almost 95% for $t_f = 1$ sec. Moreover, a very small fitness evaluation time tends to favor an intermediate number of server cycles, while a large evaluation time makes less difference. Figure 2d shows that the predicted speedups for 100 processors fit the experimental results very well.

The effect of latency is shown in Figure 2e, again for 100 processors. Latency tends to inhibit scalability when it becomes large. Moreover, it clearly tends to favor a lower number of cycles. The effect of transmission time is also to reduce scalability, as shown in Figure 2f, but a higher transmission time tends to favor a higher number of cycles.

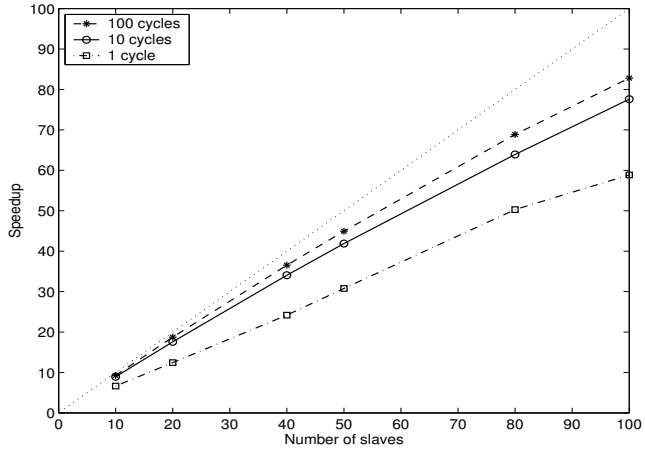

Fig. 3. Experimental speedup curves for 1, 10, and 100 server cycles in the presence of a single soft failure. Other parameters are the same as those of Figure 2.

Previously, it was assumed that slave processors would always provide a fitness evaluation value for all individuals that they receive. This might be realistic in the context of a dedicated Beowulf cluster, but it becomes a very strong assumption in the context of a LAN of multipurpose workstations that might become unavailable sporadically. Two types of failures can be considered here: *soft failures* where a slave returns only a subset of answers, indicating to the server that it must now sign-off (for whatever reason), and *hard failures* where it returns no answer, never reconnects, or reconnects as a new slave (i.e. because of network problems, system crash or reboot, etc.). Figure 3 gives speedup curves for the case of a single soft failure where the faulting slave sends a sign-off message to the server after evaluating 50% of its individuals. He then exits and the simulation continues with $p - 1$ slaves. These curves show that a single server cycle is a very bad policy in the presence of failures, because it essentially increases the number of cycles by one which can be equivalent to doubling the execution time. In this case, it adds half of a cycle for one slave, which can amount to a 33% drop in speedup. In practice, using 100 slave processors, we observe a drop from 81% to 59% for $\mu = n/p$. Using $\mu = n/100p$, the drop is only from 83% to 82%, approximately. Hard failure results are not given here, but conclusions are mostly the same, only with worse speedup curves.

V. System design

This section presents a quick overview of Distributed BEAGLE[1], our implementation of the master-slave architecture, which was used to validate the proposed mathematical model. This system is mostly a transparent extension of the Open BEAGLE C++ EC framework [4], which allows both the development of new EC flavors, using a generic layer of abstract classes, and the rapid deployment of classic paradigms like genetic algorithms, genetic programming, and evolutionary

[1] `http://beagle.gel.ulaval.ca/distributed`

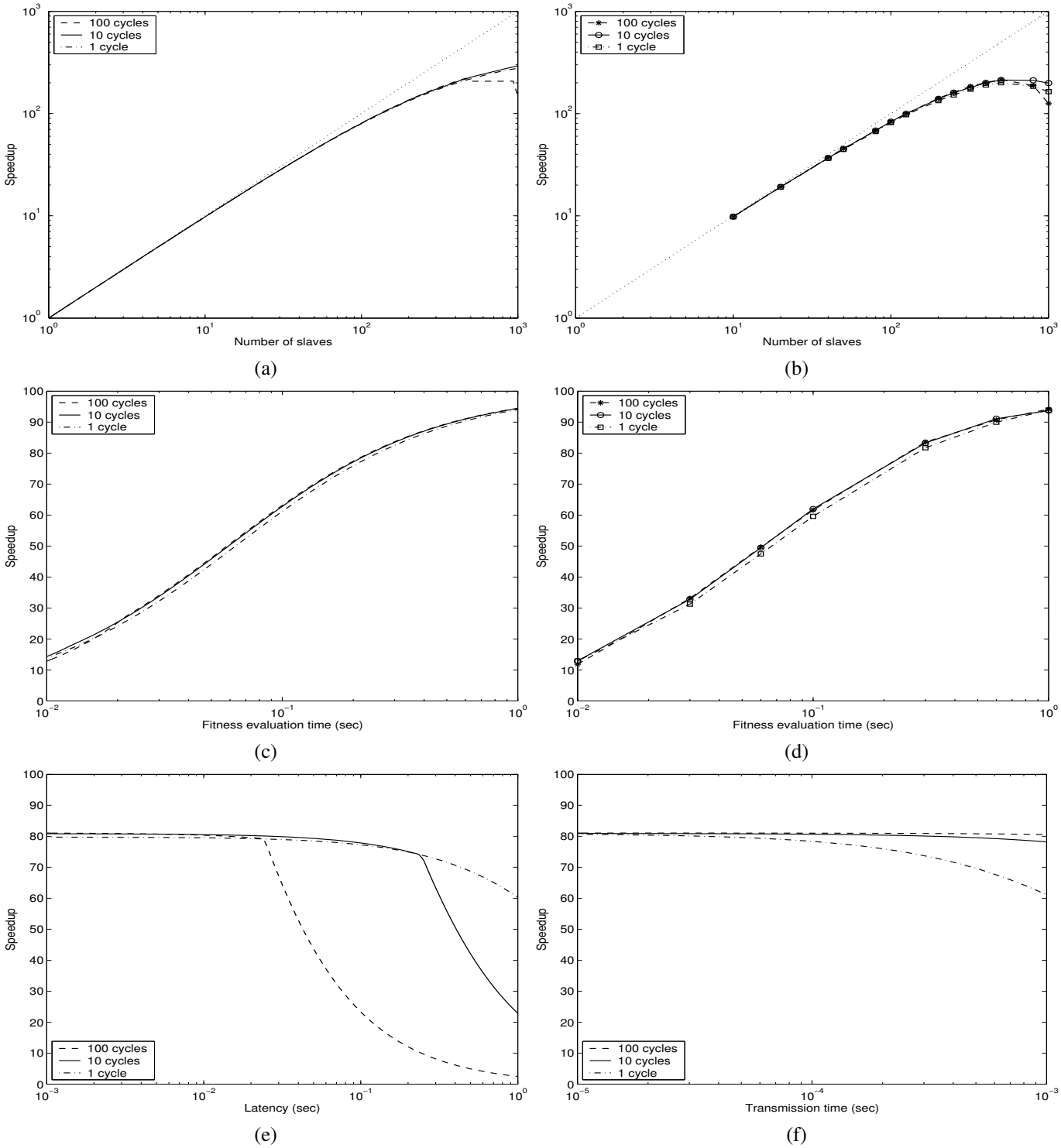

Fig. 2. Speedup curves for 1, 10, and 100 server cycles ($\mu = \{\lceil n/p \rceil, \lceil n/10p \rceil, \lceil n/100p \rceil\}$): (a) theoretical effect of p; (b) experimental effect of p; (c) theoretical effect of t_f (100 slaves); (d) experimental effect of t_f (100 slaves); (e) theoretical effect of t_l (100 slaves); (f) theoretical effect of t_{xi} (100 slaves). Other parameters are $n = 100\,000$, $t_f = 0.25$ s, $t_{xi} = 4.54 \times 10^{-5}$ s, $t_{xf} = 1.52 \times 10^{-5}$ s, $t_{mi} = 9.5 \times 10^{-6}$ s, $t_{ui} = 9.6 \times 10^{-5}$ s, $t_{mf} = 6.5 \times 10^{-6}$ s, $t_{uf} = 6.45 \times 10^{-5}$ s and $t_l = 0.001$ s

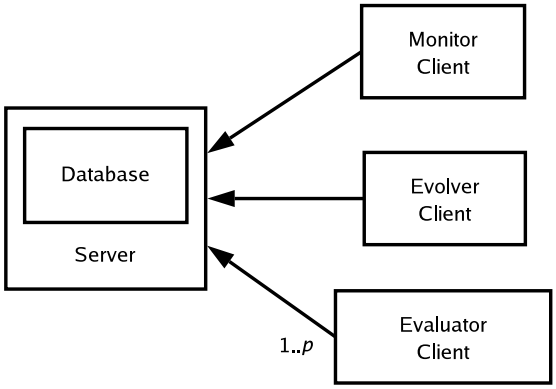

Fig. 4. Distributed BEAGLE is a client/server Architecture.

strategies, through a set of specialized layers. It also supports multi-objective optimization and co-evolution.

Illustrated in Figure 4, the client-server architecture is composed of five main module types: a *database*, a *server*, one or more *evolver clients*, and a pool of *evaluation clients*. The system works on data by separating the EC generation concept into two distinct steps: deme evolution and fitness evaluation. Deme evolution is carried out by evolver clients. It consists in applying genetic and natural selection operations to evolve the deme through one generation. Fitness evaluation is then computed by the pool of evaluator clients. When all individuals have been evaluated, the generation is finished and the demes are ready to be evolved again.

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 last 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 by the distributed and persistent evolutionary algorithm design pattern [21].

The server (master) acts as an interface between the different clients (slaves) and the database. The primary function of the server is to dispatch the demes to evolver clients, and the individuals to evaluator clients. The number of individuals sent to an evaluator client depends on a load balancing mechanism. The mechanism dynamically adjusts the number of individuals sent to each evaluator 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 evaluator client sends requests to the server for individuals that need to be evaluated. The evaluator clients are also 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 database content.

The load balancing policy is to regulate the size of individual sets in order to approximately achieve constant time periods between successive evaluator requests. 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. 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 time needed to complete a generation and assures general fault tolerance for the system.

Distributed BEAGLE is a cross-platform Unix/Windows library, developed using an SQLite database[2], portable TCP/IP socket and threading classes[3], and XML (eXtensible Markup Language) [22] for data encoding.

VI. Conclusion

This paper has introduced a mathematical model for predicting the speedup of the master-slave architecture for PDEC. This model takes into account the number of slave processors, the average network latency, the average marshalling times, the average effective network bandwidth (transmission times), and the average fitness evaluation time for individuals. Results have shown that the master-slave architecture is able to scale up well for a wide range of applications, as long as fitness evaluation time is much greater than transmission time of individuals (which is usually the case for real-world problems), and that latency is not too large. For example, an application that requires about 0.25 sec for fitness evaluation will be able to reach around 82% of the optimal speedup when using 100 processors over a typical switched LAN of 100Mbits/sec, whereas a fitness evaluation time of 1 sec yields about 95%.

The theoretical predictions made by this model have also been confirmed by an efficient and robust master-slave implementation named Distributed BEAGLE. Results have shown that the speedups predicted by the model fit very well with the actual speedups observed with this implementation. We conclude that the popular impression that master-slave PDEC cannot scale up well because of the bottleneck created by the master is misconceived. Of course this bottleneck exists. But it is not that restrictive in practice. Today's gigabit Ethernet, which is becoming mainstream in PCs, promises greater efficiency and scale. InfiniBand (10 Gbits/sec) which is just around the corner, holds even greater potential for very large scale master-slave PDECs.

Another contribution of this paper is to demonstrate that the basic policy of p-slaves-p-sets for the master-slave is sub-optimal, especially in the presence of failures, when work has to be re-dispatched to other processors.

The master-slave benefits were also put into perspective with those of the island-model. We argue that the former can be made just as robust as the latter, if not more, by assuring data integrity using a centralized persistent database. Moreover,

[2]http://www.sqlite.org
[3]http://www.gel.ulaval.ca/~parizeau/PACC

the set-up and management of a large scale master-slave can be made almost transparent, whereas the management of an island-model seems to require a more involved contribution from the user, especially if the available computer resources are not within a controlled environment (i.e. Beowulf cluster). The master-slave can efficiently exploit new resources that become available at any time and can also be made fault-tolerant to lost resources. In that context, we are currently developing a special screen-saver that will soon be installed on all of our Windows workstations which run idle most of the day. Statistics over a one week period have shown that the 55 workstations in our lab run completely idle 78% of the time. An important challenge, however, is to make this screen-saver secure enough so as to prevent the distribution of viruses in place of evolutionary computations!

REFERENCES

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

[2] J. R. Koza, F. H. Bennett III, D. Andre, and M. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufman, 1999.

[3] J. Koza, M. Keane, M. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

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

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

[6] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, October 2002.

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

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

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

[10] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, and A. Chircop, "ECJ 13: A java-based evolutionary computation and genetic programming research system," Seen July 20, 2005 at http://cs.gmu.edu/~eclab/projects/ecj, 2005.

[11] S. Cahon, N. Melab, and E. Talbi, "ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics," *Journal of Heuristics*, pp. 357–380, 2004.

[12] J. Costa, N. Lopes, and P. Silva, "JDEAL: The java distributed evolutionary algorithms library," Seen July 20, 2005 at http://laseeb.isr.ist.utl.pt/sw/jdeal/home.html, 2004.

[13] B. Punch and E. Goodman, "Galopps 3.2," Seen July 20, 2005 at http://garage.cps.msu.edu/software/galopps, 2002.

[14] K. C. Tan, A. Tay, and J. Cai, "Design and implementation of a distributed evolutionary computing software," *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, vol. 33, no. 3, pp. 325–338, August 2003.

[15] F. Fernandez, M. Tomassini, and L. Vanneschi, "An empirical study of multipopulation genetic programming," *Genetic Programming and Evolvable Machines*, vol. 4, no. 1, pp. 21–51, 2003.

[16] 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*, vol. 2. Orlando, Florida, USA: Morgan Kaufmann, July 1999, pp. 1484–1490.

[17] J. Koza, L. Jones, M. Keane, M. Matthew, and S. Al-Sakran, "Toward automated design of industrial-strength analog circuits by means of genetic programming," in *Genetic Programming Theory and Practice II*, U. O'Reilly, R. Riolo, G. Yu, and W. Worzel, Eds. Kluwer Academic Publishers, 2004, ch. 8, pp. 121–142.

[18] C. Gagné, M. Parizeau, and M. Dubreuil, "A robust master-slave distribution architecture for evolutionary computations," in *Late Breaking Papers of GECCO 2003*, Chicago, IL, USA, July 2003.

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

[20] H. Pierreval and J.-L. Paris, "Distributed evolutionary algorithms for simulation optimization," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 30, no. 1, pp. 15–24, January 2000.

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

[22] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0 - W3C recommendation 10-february-1998," Available on the Web at http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, Tech. Rep. REC-xml-19980210, 1998.