Galapagos: A Generic Distributed Parallel Genetic Algorithm Development Platform for Computationally Demanding ITS Optimization Problems

Nicolas Kruchten, Research Assistant

Baher Abdulhai, Associate Professor and Director

David de Koning, Research Assistant

Intelligent Transportation Systems Centre and Testbed

Department of Civil Engineering

University of Toronto, Toronto, Ontario, Canada M5S 1A4

Tel/Fax: 416-946-7662/416-978-5054

baher@ecf.utoronto.ca

(Words = 5471, Figures and Tables = 8, Total word equivalent = 7471)

Submitted for Presentation and Publication

TRB 2004

ABSTRACT

Numerous Intelligent Transportation Systems (ITS) problems can be formulated as optimization problems. One promising approach to solving such optimization problems is genetic algorithms. Due the fact that most transportation optimization problems are large in size (i.e. have many variables) and computationally very demanding, their solution can be very or even prohibitively slow. This paper introduces our new experimental development framework for distributed parallel genetic algorithms, which we call Galapagos. The main motivation behind the development of Galapagos is to speed up the convergence to solution of large scale ITS problems to usable, practical timescales.

In highlighting the effective usage of genetic algorithms to solve difficult problems in ITS, we describe advanced genetic algorithms using parallel population structures and distributed computation, then describe combinations of these approaches and highlight key issues in terms of design and implementation, as well as benefits in terms of speed gains. Galapagos components are then described, and lastly a case study on calibrating an incident detection algorithm is presented to give one illustration of how Galapagos allows for very rapid convergence. The case study covers a system previously developed in-house called Genetic Adaptive Incident Detection (GAID), which uses a single-population single-computer genetic algorithm to optimize sixteen parameters of a probabilistic neural network classifier. It is shown that recasting GAID as a Galapagos application significantly and predictably sped up the convergence of this algorithm, suggesting that the same could be done for a variety of GAs in the ITS field.

INTRODUCTION

Many Intelligent Transportation Systems (ITS) problems can be formulated as optimization problems, that is, finding the vector **x** such that a given function z(**x**) is minimized. These optimization problems either have large numbers of variables or are simply intractable by traditional optimization techniques such as gradient-based algorithms, which tend to find local optima, rather than converging to the desired global optimum, when they are usable at all. Some functions needing to be optimized are non-differentiable, ruling out traditional search methods. A representative example of this would be model-parameter calibration by minimization of an error term expressed as a function of these parameters. In this case, **x** is the vector of parameters, and z (**x**) is the error term: calibration of the system requires the minimization of this error term. Examples of such systems can be traffic simulation models, neural network classifiers and origin-destination-estimators, to name a few (*1, 2*).

Most recently, evolutionary algorithms, specifically *genetic algorithms* (GA) have emerged as promising solution methods which are not only able to tackle such difficult problems, but also able to find global optima or at least better local optima (*3*). Genetic algorithms, however, can be impractically slow. For example, GENOSIM, a genetic algorithm based optimizer for large scale microscopic traffic simulation models that was developed in-house at the University of Toronto, takes one full week to optimize a 200 km^2 simulation network on a typical Pentium 3 1.0 GHz PC. Such calibration is to be repeated quite often: every time a significant number of changes are introduced to the network. Another example is the problem of dynamic origin destination estimation, which plays a central role in our research. This problem can be cast as an optimization problem with a very large number of variables for a not-too-large transportation network, but must be solved within a minute or two, which is impossible on any single desktop computer.

Thus, we sought an expedient approach to the optimization of such demanding ITS applications. In this paper we introduce our new experimental development framework for distributed and parallel genetic algorithms, which we call Galapagos. The main motivation behind the development of Galapagos is to speed up the convergence to solution of large scale ITS problems to usable, practical timescales.

The first section of this paper is a concise introduction to genetic algorithms. The next two sections treat GA parallelization and distributed computing separately, starting with parallelization. The application of distributed computing to GAs and parallel GAs is described next. The last two sections describe Galapagos, a new software platform developed to study and use distributed and parallel genetic algorithms in the context of ITS problems specifically. An application of Galapagos to the calibration of an incident detection algorithm is presented

GENETIC ALGORITHMS

Genetic Algorithms (GAs) are stochastic search methods based on the principles and mechanisms of natural selection and ‘survival of the fittest’ from natural evolution. Since their introduction in the 1970s, in Holland’s study of adaptation in artificial and natural systems (*3*), GAs have become popular optimization methods. By simulating natural evolutionary processes, a GA can search the problem's solution space thoroughly and multi-directionally by maintaining a population of potential solutions and encouraging exchange between these directions. The population undergoes a simulated evolution: at each generation the relatively "good" solutions reproduce, while the relatively "bad" solutions die. To distinguish between different solutions, an objective function is used for evaluation that plays the role of environment.

Mathematically, a GA emulates the concepts of competition and selection in populations of living creatures to generate a good solution vector, **x,** to the problem of minimizing a given function, z(**x**). Each solution is treated as an individual or *chromosome*, the components of which are *genes.* Each chromosome has a *fitness,* usually equal to -z(**x**) in a minimization context. A variety of genetic *operators* (selection, mutation, crossover) are repeatedly applied to an initial population of such individuals, which can be randomly generated or 'seeded', in order to find fitter and fitter individuals, that is, solutions **x** with lower and lower values of z(**x**).

FIGURE 1 Generalized genetic algorithm flowchart.

There are many different ways to implement a GA for a given problem, with variations in the representation of the solution and in the choice of genetic operators. Each chromosome encodes a particular solution to the problem at hand, usually a vector of numbers. These numbers can be converted into one long binary string or can be used as real values. The various genetic operators chosen for each step in Figure 1 also define the type of genetic algorithm. At each iteration of the GA, new chromosomes are generated from the current population P(t), populating the next *generation,* P'(t), by *mutating* a single parent and/or by combining genes or portions of the solution from two or more parents via a process known as mating, or *crossover.*

Which chromosomes are to be used to produce the new generation depends on the operator chosen, as does the manner in which each generation is integrated into the population. For instance, in GAs implementing *elitism*, each new batch of chromosomes is compared against their parents and the best members stay in the population, these are known as *crowding* GAs. Another approach is to have all of the new generation of chromosomes enter the population, completely replacing the old one. The number of chromosomes in the population and in each generation is also a variable that will impact the GA's operation. One notable configuration is the *steady-state* GA, where after each generation is evaluated, only one chromosome is generated and then inserted into the population only if it displaces another chromosome. Certain GAs also include *adaptive* operators and parameters, where, for example, the rate at which mutation occurs will change as the chromosomes get fitter. A wide variety of parameters and operators have been proposed and analyzed to varying degrees in the literature (*4*) while still fitting into the generalized flowchart above, but it is important to note that no single configuration has proven to work best in all cases. The choice of operators and parameters is more of an intuitive activity than a science, but guidelines and heuristics have been developed (*5*).

PARALLEL GENETIC ALGORITHMS

More sophisticated GA variants, *parallel genetic algorithms* (PGA), which have multiple interacting sub-populations (or *demes*), have been used to generate good solutions more efficiently. Single-deme GAs are also known as *panmictic* GAs, because every chromosome in the population can mate with every other chromosome in the same population, which is not true of PGAs, where mating is restricted to chromosomes within the same deme. Panmictic genetic algorithms can generate good solutions to optimization problems, but are not entirely immune to entrapment in local optima. By using multiple interacting sub-populations, GAs have not only been shown to be less prone to getting trapped in local minima, as multiple areas of the search space are explored independently, but also to converge to a solution more rapidly *(6*).

A PGA, like a panmictic GA, mimics the situations found in nature: semi-isolated islands where groups of individuals can evolve independently, with sporadic migrations (this is exactly the type of situation Darwin went to the Galapagos Islands to study in bird populations). On any given island, the population can begin to lose diversity, or become trapped in a local minimum, but this can be alleviated by the insertion of new genetic material from another island. In this way, the global population maintains its genetic diversity and each deme (island) can explore a different area of the solution space, with periodic input of genetic material from other demes.

Each deme in a basic PGA is essentially a separate GA in its own right. The various demes interact through a process known as *migration*, where some chromosomes are periodically sent or copied into another population. This period is called the *epoch*. There are various ways to define the duration of an epoch, the selection of migrants, whether they are copied or moved, and which chromosomes of the destination deme they replace. Another, very important feature of PGAs is the *topography* used to link the demes.

The topography defines which demes send migrants to which other demes. In a fully connected topography, every path is possible, but this is often not the optimal configuration. Various topographies have been proposed and studied, but two major classes are important to note: fine-grained topographies, also known as diffusion (*5, 6, 7*) and coarse-grained topographies, also known as cellular GAs. *Granularity* refers to the ratio of computation time to communication time.

In a coarse-grained PGA, there are usually a few, highly connected large demes and the migration rate is low. This low migration rate means that more time is spent computing values and applying genetic operators than migrating chromosomes. In fine-grained PGAs, there are usually a large number of loosely connected, small demes with a high migration rate. The choice of topography for a given problem is beyond the scope of this paper, but has been explored in the literature (*5, 6, 7*).

FIGURE 2 Possible PGA topographies: (a) fine-grained and (b) coarse-grained.

One major drawback common to all GAs, however, is that they often require hundreds or thousands of function evaluations: for every **x**, z(**x**) needs to be computed, which can be very time-consuming, especially in model-parameter calibration problems, where evaluating the objective function can amount to running a whole simulation.

DISTRIBUTED COMPUTING

When faced with an excessively computationally intensive problem, the usual solution is to use a faster computer (i.e. with a faster processor). More recently, however, affordable computers with multiple processors, cheap computer clusters and *'grid computing'* (*8*) have also emerged as effective ways of dealing with computationally demanding tasks and together they are known as distributed, or parallel, computing. Each of these systems includes more than one processor, either in the same machine or distributed over a network.

A processor is inherently sequential; it cannot execute two programs or *processes* simultaneously. This is often emulated by multiplexing, however, with each process being allowed to run in turn for a short amount of time, giving the impression of simultaneity (pseudo-parallelism). Using more than one processor means that more than one process or more than one part of the same process can execute truly simultaneously with no loss in speed. Multi-processor machines have been dropping in cost since their inception and consumers can now readily obtain very powerful dual-processor machines at an accessible price. Computer clusters are groups of separate computers, with single or multiple processors, linked together and specially programmed to work cooperatively to solve problems. The concept of 'grid computing' is based on the idea of having a large, fluid pool of available computing resources, counted in the hundreds or thousands of computers that can be harnessed on demand by a resource-hungry process.

One example of this is the very successful SETI@Home project (*9*), in which members of the public can download software that runs while their computer would otherwise be idle. Each computer processes blocks of radio-telescope data to help in the search for extra-terrestrial intelligence (SETI). Grid components can be added to or removed from the pool as appropriate and computation continues. A simple way of dealing with this sort of system is to have one central *dispatcher* which continually gathers *jobs* from *master* processes, distributes them among the available *slave* processes in the pool, gathers the results of the jobs' execution and returns them to the masters. In this fashion, the dispatcher acts as an abstraction layer between the masters and the slaves: there can be as many of each as needed or available, and neither must deal directly with the other.

FIGURE 3 Simple grid-computing architecture.

Note an important distinction between the 'parallel' in PGA and what is described above and known as 'parallel computing'. Parallel computing distributes the computation over multiple processors, all executing simultaneously (i.e. in parallel); whereas, the 'parallel' in PGA usually refers to the structure of the population. A PGA can execute sequentially on a single processor, and the parallelism in the evolution of the demes is simulated. *By convention in this research, we use the term 'distributed' instead of 'parallel' when referring to computation, and use 'parallel' only when referring to the GA's population structure.*

Not all problems can be effectively distributed, and a good measure of this is the fraction of the problem (in terms of execution time) which can be parallelized in time, as the portions of the algorithm which must run serially cannot be parallelized. In most GAs, the parallelizable step is the most time-consuming one: the evaluation of the chromosomes. Thus there is a large enough distributable fraction that GAs are known to be ‘embarrassingly parallelizable’.

In the distribution of computational tasks, the following merit consideration:

- *Speedup*: the ratio of execution speed of n processors versus 1 processor. (The ideal result in distributed computing is linear speedup: 2 processors execute in half the time, 3 in a third and so on.)
- *Efficiency*: the fraction of linear speedup achieved.
- *Scalability*: how speedup changes with n.
- *Robustness*: how well the system recovers from errors (e.g. processors crashing mid-computation, etc.)
- *Adaptability*: how well the system can integrate heterogeneous resources, that is, processors of different speed and memory characteristics or network connection qualities.
- *Transferability*: how easy is it to run the system in a totally different environment (related to adaptability).

The possibility of super-linear speedup (i.e. efficiencies higher than 1) is a controversial issue. Common-sense suggests that any task that can be parallelized can be run serially as, if not more, efficiently due to parallelization costs but some have suggested various justifications for the existence of super-linear speedup, including hidden costs in serial computing (*10*). Great care must be taken in evaluating speedup in various distributed applications, as various factors could lead to mistakenly observing super-linear speedup. Speedup is also a difficult measure in distributed systems with heterogeneous processors, and other measures may be more appropriate in these contexts (*6*).

Scalability, robustness and adaptability are important to varying degrees, depending on the application. If the system is to be deployed on a particular, well-known homogeneous system with a fixed number of identical processors, as in the case of a given multi-processor computer, then scaling, flexibility and transferability are less important. If, however, the system is to be used on a variety of networks, where processors are prone to crashing or the number or capacity of processors is known to be variable (e.g., in a grid), these issues take on more importance.

COMBINING DISTRIBUTED COMPUTING AND GENETIC ALGORITHMS

Distributed computing can be used with GAs in a variety of ways, in order to take advantage of problem-specific properties or the available hardware. While any GA, including PGAs, can be implemented and run on a single processor, they are easy to distribute and this can greatly speed up convergence. The function evaluation step of a GA (see figure 1) is inherently easy to distribute; many such evaluations are required and they do not depend on one another, so they can be executed at the same time on different processors. For a panmictic GA, distribution takes the form of a master/slave architecture, where all of the genetic operations happen within the master process and all of the function evaluations occur within the slave processes. This type of architecture is a canonical application of grid computing as described above.

PGAs can be very effectively distributed, with one processor managing each deme. In this case, there are no masters or slaves and the processors are known as *peers*. The problems of how to structure a GA's population and how to distribute the computational load are thus not unrelated: the available configuration of computer and network hardware can impact the choice of GA structure (PGA vs. panmictic, fine-grained vs. coarse-grained, etc) as can the nature of the problem. If the definition of granularity seems out of place when discussing a PGA's population structure, it is because this term comes from the distributed computing field, where communication time between processors is an important dimension to consider when designing systems. If the *cost* of a single evaluation a – its duration in time – is a tenth of the cost of communication to send the results somewhere, then it might be more efficient to perform 20 to 30 evaluations before sending any data, if possible. With this in mind, it becomes clear what types of hardware configurations or problems are well suited for coarse or fine-grained distribution PGA architectures. When the cost of computation of one work-unit is higher than the cost of communication, a finer-grained architecture might be appropriate. When communication costs are comparatively high, it makes more sense to use a coarser-grained distribution scheme.

Beyond the basic distributed PGA architecture, more complex forms are possible. Each deme in a PGA is essentially a GA, and could in fact be another PGA with a different granularity or a distributed panmictic GA. Demes can also be substantially different from each other in a given PGA. PGAs where demes are not identical (i.e. different operators, parameters or population structure) are known as *heterogeneous* PGAs. Super-linear speedup has been reported for both homogeneous and heterogeneous PGAs, but as noted, such results are controversial (*6*).

GA design is mostly concerned with operators and parameters while PGA design addresses migration and topography. Distributed computing is a field in its own right, as it is possible (but often much harder) to distribute other algorithms than GAs. The integrated design of distributed PGAs, however, incorporates all of the above challenges as well as that of mapping computational tasks to specific hardware: the end result must be an efficient GA whose operators, parameters and population architecture are well-suited to both the problem at hand and the hardware and network available.

GALAPAGOS AND LIGHTGRID

We have designed and built Galapagos and LightGrid to implement distributed PGAs primarily for ITS applications. Galapagos is a generic distributed PGA development platform, built on top of a lightweight, generic grid-computing platform we name LightGrid (see Fig. 4), both of which were concurrently developed at the University of Toronto ITS Centre. Galapagos makes it easy to design, implement and use any of the distributed and parallel GA configurations described in this paper, tailored to the problem and available hardware.

FIGURE 4 Layered architecture of LightGrid and Galapagos.

This layered approach to software development allows replacement of any of the layers by other software that performs the same function. For example, we could replace the LightGrid layer by a more heavy-duty grid-computing engine, or Galapagos can run in a 'single-computer' mode, bypassing the LightGrid layer altogether. The Galapagos and LightGrid components can also each be upgraded independently of the other, while maintaining functionality of older applications.

The LightGrid software consists of a simple master/slave model mediated by a job-dispatcher process. A LightGrid master process assigns a job to the dispatcher, which queues it for processing by a LightGrid slave process. This system is a 'light-weight' grid-computing platform as it is intended for controlled use in an academic environment. Using LightGrid, a variety of non-GA distributed applications can also be easily developed.

Galapagos extends the LightGrid master and slave processes by adding generic GA capability. A Galapagos slave process essentially is an objective function evaluator, while a Galapagos master manages a given deme or set of demes. Galapagos is a generic system because it is not problem-specific: it provides program building blocks with which to implement any GA in any configuration. All that is needed to run any given GA configuration is the provision by the developer of 5 modules:

- *a selector*: this module defines the parent-selection step in a GA iteration as well as the rules governing which of the new chromosomes enter the population
- *a breeder*: this module defines the type of crossover used in a GA iteration
- *a mutator*: this module defines the rules for gene mutation
- *a migrator*: this module defines PGA topography, and migration selection rules
- *an evaluator*: this module is deployed as part of the slave process and defines the objective function. This module can invoke other third-party software to perform specific computation (for instance, can invoke a traffic simulator to run a network under certain control configuration).

FIGURE 5 Galapagos flowchart illustrating usage of modules.

This highly modular system was implemented according to the principles of object-oriented (OO) programming with the Java programming language. OO programming embodies the idea that each piece of data in a program is an object, which has both associated attributes (variables) and operations (functions or procedures) (*11*). This approach leads to a clear encapsulation of data into re-usable and understandable software components. Java was designed specifically for OO programming and is well suited to this sort of application. As well, Java software is portable; it can run on almost every operating system in existence, meaning that the software developed is adaptable and transferable to computers running a variety of operating systems such as Windows, Linux, and other UNIX variants.

Galapagos allows the GA developer to specify any number of populations, using any GA operator, in any deme topography, distributed over separate processors in a variety of ways. A library of selector, breeder, migrator and mutator modules is under development, allowing a developer to simply pick and choose operators and parameters, but the system is open to extension (i.e. anyone can write a new operator module and use it). The evaluator module defines the problem itself: the objective function. The evaluation of the objective function can be written in Java, but it is also possible for this module to invoke third-party programs; for example, if the problem is model-parameter-calibration for a given software package, the evaluator can run this software package with a given set of parameters (the chromosome **x**) and score the output in Java according to some scheme (the objective function z(**x**)).

GALAPAGOS IN ACTION ON INCIDENT DETECTION

Genetic algorithms have been used in the past to perform automated incident detection by calibrating an artificial neural-network classifier (*2*). We used Galapagos to distribute and parallelize such a system, Genetic Adaptive Incident Detection (GAID). The GAID system comprises of a standard genetic algorithm whose 16-gene chromosomes represent the 16 smoothing parameters for a probabilistic neural-network (PNN) with 16 traffic measurements. The objective function consists of 'scoring' the PNN's ability to correctly classify incidents and non-incidents. This 16-dimensional function is difficult to differentiate and standard optimization techniques are not adequate to optimize the objective function. Further details about GAID can be found in (*2*), which does not specify the exact operators or parameters to be used in implementing the system. The implementation of GAID used in this research used the following operators and parameters, for reference: a single population of 50 chromosomes, generations of 50 chromosomes each, random selection of 2 parents, discrete recombination crossover, worst bias replacement crowding and adaptive mutation rate linearly varying with the fitness range in the population.

The hardware available to do this consists of a large number of moderately powerful single-processor desktop machines: 50 computers, running at 1.4 GHz and using the Linux operating system which are a part of the University of Toronto's Engineering Computing Facility (ECF) public computer labs. These machines are linked together into a standard megabit Ethernet network switch where every computer can easily communicate with every other one. The Galapagos slaves were run as background processes on these publicly accessible machines while normal usage of these computers by engineering students continued undisturbed.

GAID was successfully distributed under a panmictic model across ECF using Galapagos. The computational load was distributed across 50 computers with no loss in solution quality compared to the single-computer version of GAID. The grid-computing nature of the LightGrid engine also proved to resist well to large changes in slave performance or instances where slave machines crashed or stopped returning results.

The speedup experienced is somewhat difficult to quantify for a few reasons. First of all, GAs, being stochastic processes, can yield a wide range of convergence times and, if they do not converge to the global optimum, can converge to a wide variety of local optima, depending on the function. Second, the tests were run on a variety of machines, under a variety of load conditions (i.e. some machines were being used concurrently by students, reducing the amount of processing power available to the Galapagos slave process). Third, speedup involves comparing the execution time of the whole system to that of one computer, and the execution speeds of individual computers varied over time and with respect to each other. Finally, the relationships between the number of slaves available, the distribution of their processing power and the number of chromosomes per generation affect the execution speed of the whole system in a complex manner (which is modeled below), as can the job-dispatching policy employed by the dispatcher (i.e. in what order the jobs are assigned and to which slave), but while that is beyond the scope of this paper, Galapagos is an ideal platform for studying these relationships.

The time to convergence can be estimated with the following reasoning: the distribution of GAID using Galapagos did not modify the panmictic algorithm, it just distributed the parallelizable step, so GAID on one or many computers should converge in the same number of generations, empirically found to be around 23 with the set of operators and parameters used. The problem, then, is estimating the amount of time a set of computer will take to process a single generation, which can be lower-bounded with the following equation:

$$t_{gen} = \min t \ni \sum_{i=1}^{S} floor\left(\int_{0}^{\frac{t}{\left(1+\frac{1}{q}\right)}} v_i(x)\,dx \right) = g$$

where:

- t_{gen} = the computation time for one generation
- s = the number of slave processes
- g = the number of chromosomes per generation
- $v_i(x)$ = the 'instantaneous computation speed' in terms of 'evaluations per time' of slave i at time x
- q = the granularity (the ratio of evaluation cost to communication cost)

The right-hand side of the equality is the number of evaluations completed at time t, thus when this is equal to g, the generation has been evaluated, assuming that the fastest slaves always get assigned jobs first, which is a non-trivial task for the dispatcher, as it would involve predicting $v_i(x)$. This model is quite complex, as it incorporates the variation over time of a given slave's computational power. Taking v_i to be constant (the inverse of the costs of evaluation plus communication), we get:

$$t_{gen} = \min t \ni \sum_{i=1}^{S} floor(v_i t) = g$$

which is much more practical, given that it is unlikely that we will ever have access to $v_i(x)$. The floor function is used in both of these because we cannot say that having two half-evaluated chromosomes is equivalent to one fully-evaluated one, which is the essence of the problem of dispatching policies. This equation also gives us an upper bound on the speedup and efficiency we can expect on the distributable portion of a GA, which comprises the bulk of the algorithm's computational load, given that the selection/mating steps in the algorithm execute in an almost negligible time compared to the evaluation of a chromosome.

The 50 computers available to us varied in speed but all took around 4 seconds per evaluation (including communication cost, which was negligible), and the following graphs show empirically observed evaluation time for a generation, speedup and efficiency of the system as compared to an ideal system comprised of identical machines that can compute and communicate the results of every job in 4 seconds and to the ideal linear speedup case. The speedup reported is with respect to the speed of the single-computer data point. The observed and predicted values are extremely close, and the predicted values correctly lower-bound the evaluation time and upper-bound the speedup and efficiency.

FIGURE 6: Generation evaluation time of distributed GAID using Galapagos.

FIGURE 7: Speedup of distributed GAID using Galapagos.

FIGURE 8: Efficiency of distributed GAID using Galapagos.

Note that in all three graphs, the observed and predicted behaviours both approach ideal linear speedup when g/s is close to an integer value (i.e. at s = 5, 6-7, 10, 12-13, 25 etc) and that the marginal gains in evaluation speed are almost zero in between these points. This is due to the fact that when, 48 machines are available, two of them must evaluate two jobs in order to make 50, while the other 46 sit idle, reducing efficiency. This scalability behaviour has important implications when designing distributed panmictic GA systems, as the generation size g is not always a variable that can be set to arbitrary values. Some experiments were done involving an asynchronous system with a generation size of 1, which displayed qualitatively good scalability and adaptability, but overall less speedup. The details of that system are outside the scope of this paper.

In principle, similar speedups and efficiencies could easily be obtained with Galapagos for GAs targeting other transportation problems, using already-available under-used hardware in existing networks (like the ECF computers), essentially allowing access to super-computer processing power with no additional outlay of funds.

GAID was also successfully implemented as a PGA using Galapagos as a proof-of-concept, but no rigorous analysis of its performance was performed beyond noting that it did not perform any worse than the panmictic version.

CONCLUSIONS

Galapagos has already successfully been used to drastically lower the computation time of GAs applied to difficult transportation problems and further applications are in development. It will be used in the near future to implement various new GA projects at the University of Toronto ITS Centre and can also be used to re-implement existing GAs as distributed GAs. Standard single-computer, single-population genetic algorithms can take a long time to generate good solutions to demanding optimization problems in ITS. However, parallelization of the GA's population structure and the distribution of computational load are very effective and easily applied ways to speed up the solution of challenging optimization problems. The Galapagos platform is a flexible, easy-to-use and easy-to-deploy distributed PGA software package and is very promising in terms of opening up new avenues of research in transportation by bringing within reach the possibility of very rapidly getting good solutions to previously intractable transportation problems.

REFERENCES

1. Ma, T., and Abdulhai, B., “GENOSIM: A Genetic Algorithm-Based Optimization Approach and Generic Tool for the Calibration of Traffic Microscopic Simulation Parameters”, Journal of the Transportation Research Record, TRR # 1800, 2002.
2. Roy, P., and Abdulhai, B., “GAID: Genetic Adaptive Incident Detection for Freeways”, Transportation Research Board 2003, Also Accepted, Journal of the Transportation Research Record, 2003 (forthcoming).
3. Holland J.H. 1975, Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, Michigan
4. Chambers, Lance. Practical Handbook of Genetic Algorithms. CRC Press, New York, 1995.
5. Cantú-Paz, Eric. Designing Efficient and Accurate Parallel Genetic Algorithms, PhD Thesis, 1994
6. Alba, E., Nebro, A., and Troya, J. Heterogeneous Computing and Parallel Genetic Algorithms. *Journal of Parallel and Distributed Computing 62*, 1362–1385 (2002)
7. Alba, E, and Troya, J. An Analysis of Synchronous and Asynchronous Parallel Distributed Genetic Algorithms with Structured and Panmictic Islands IPDPS 1999 Workshop Online Proceedings http://ipdps.eece.unm.edu/1999/biosp3/alba.pdf Access July 30, 2003
8. Myer, Thomas, IBM, May 2003 http://www-106.ibm.com/developerworks/grid/library/gr-fly.html?ca=dgr-lnxw01GridFlyover Accessed July 30, 2003.
9. SETI@Home http://setiathome.ssl.berkeley.edu/ Accessed July 30, 2003.
10. Gustafson, J. "Fixed Time, Tiered Memory, and Superlinear Speedup”. *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, October 1990.
11. Booch, Grady. *Object-oriented analysis and design with applications*. Addison-Wesley, Menlo Park, CA, 1994.

LIST OF FIGURES

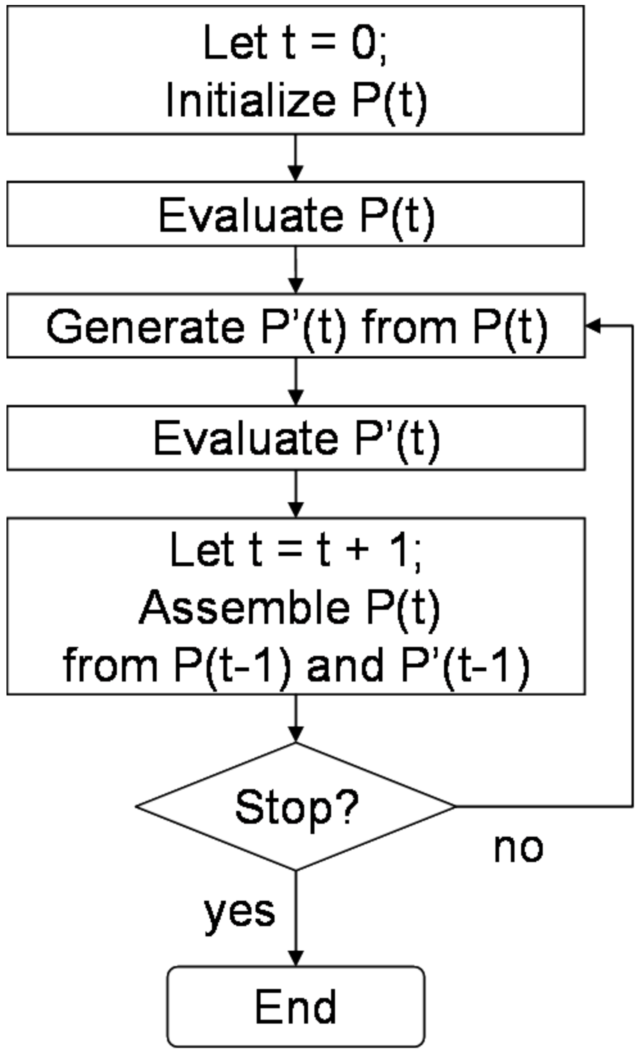

FIGURE 1 Generalized genetic algorithm flowchart.

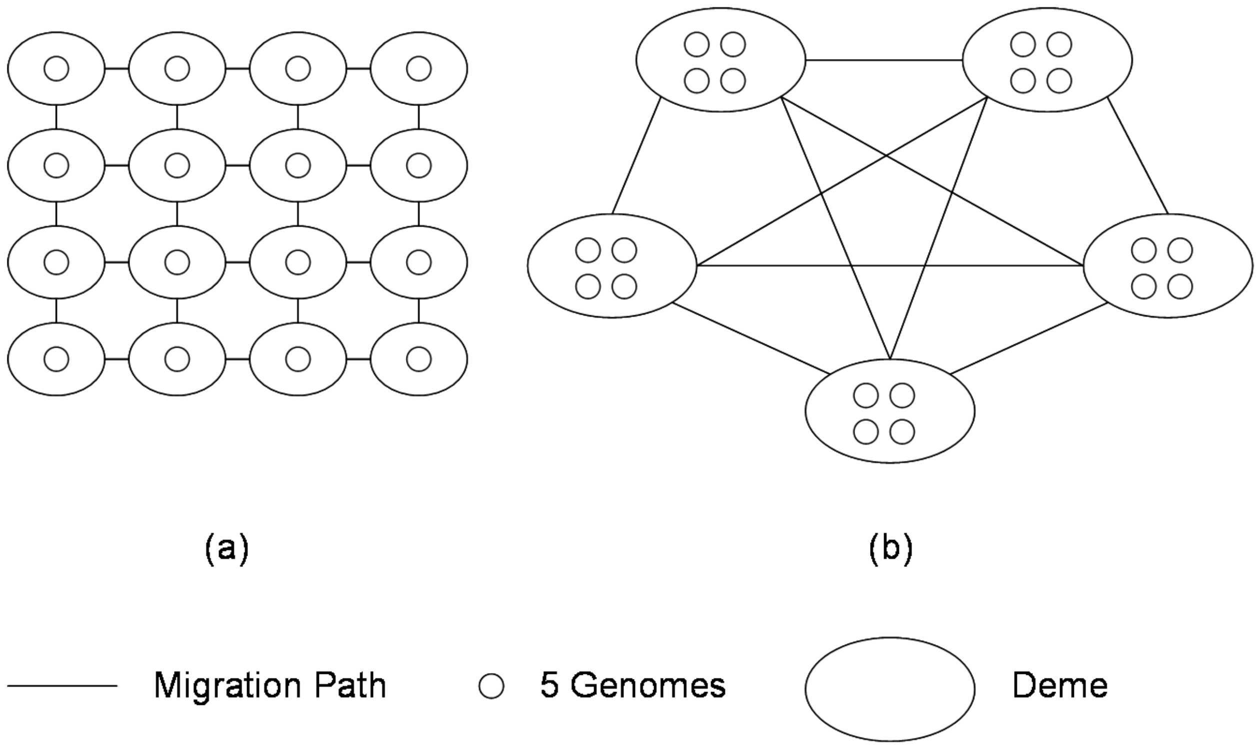

FIGURE 2 Possible PGA topographies: (a) fine-grained and (b) coarse-grained.

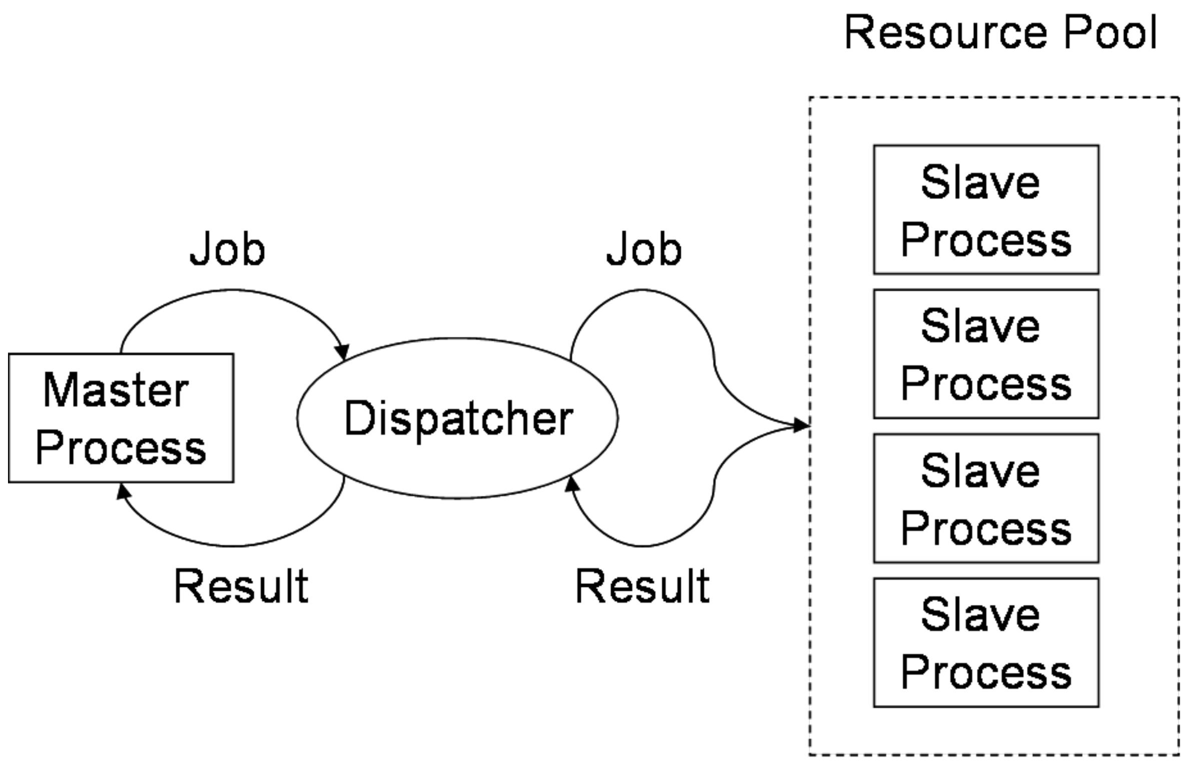

FIGURE 3 Simple grid-computing architecture.

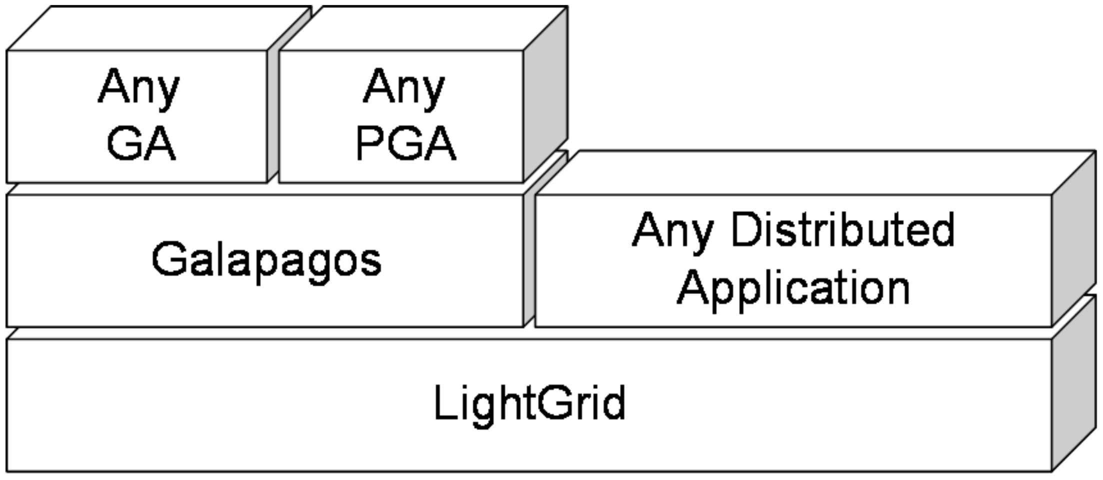

FIGURE 4 Layered architecture of LightGrid and Galapagos.

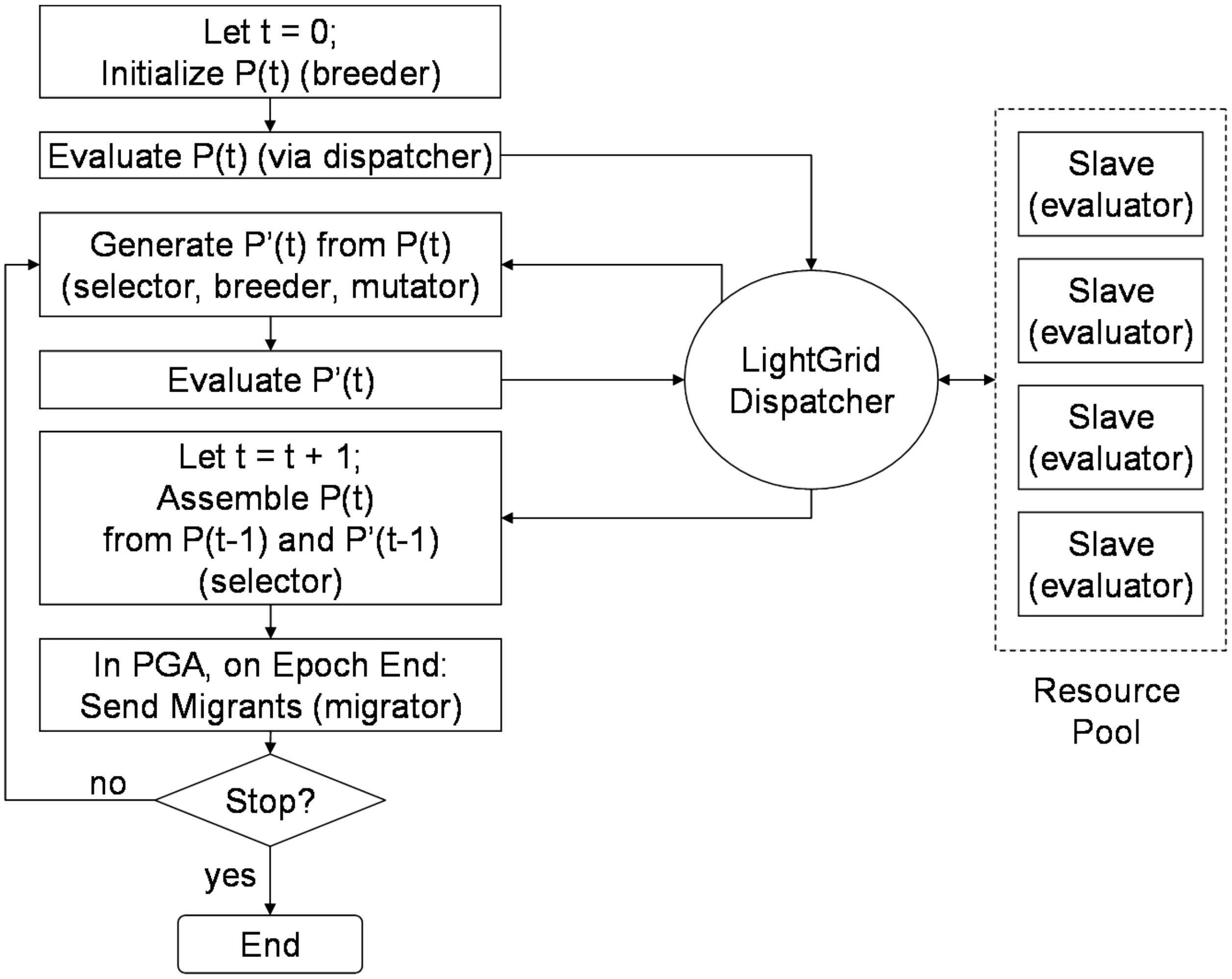

FIGURE 5 Galapagos flowchart illustrating usage of modules.

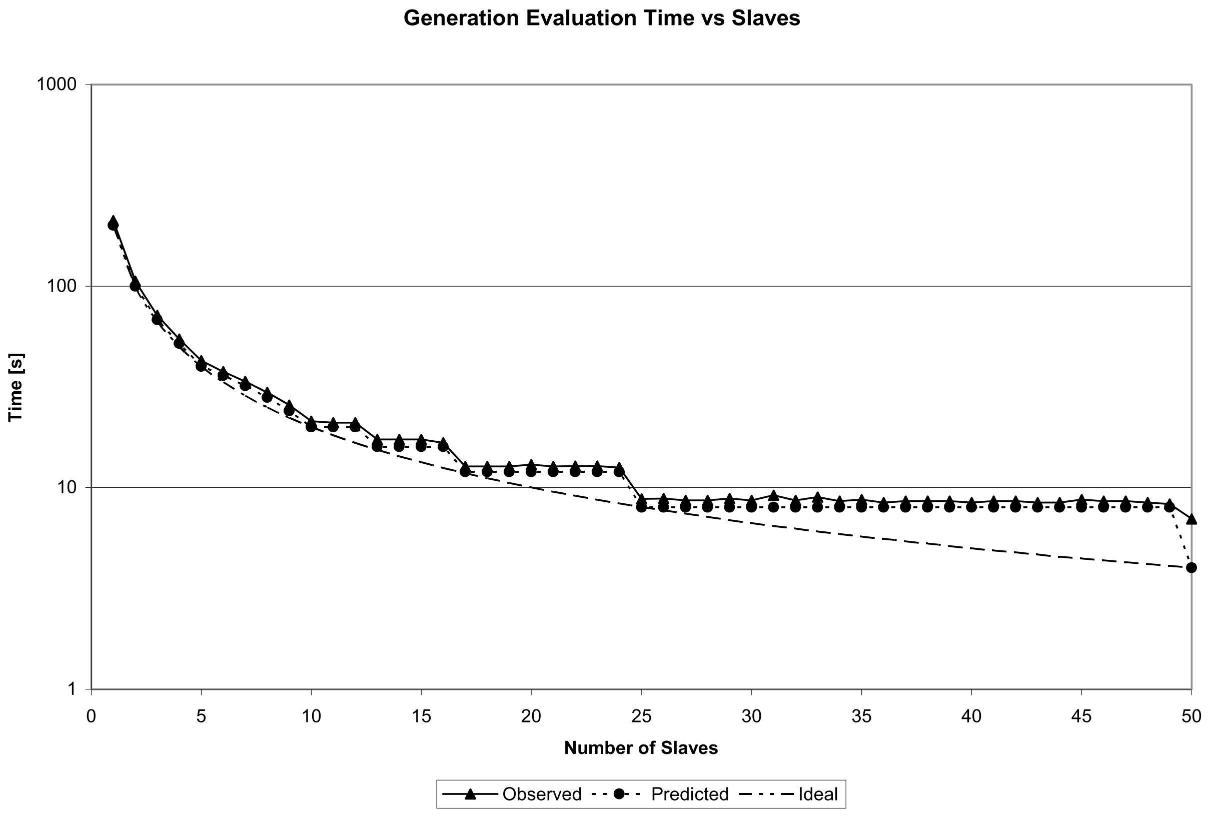

FIGURE 6: Generation evaluation time of distributed GAID using Galapagos.

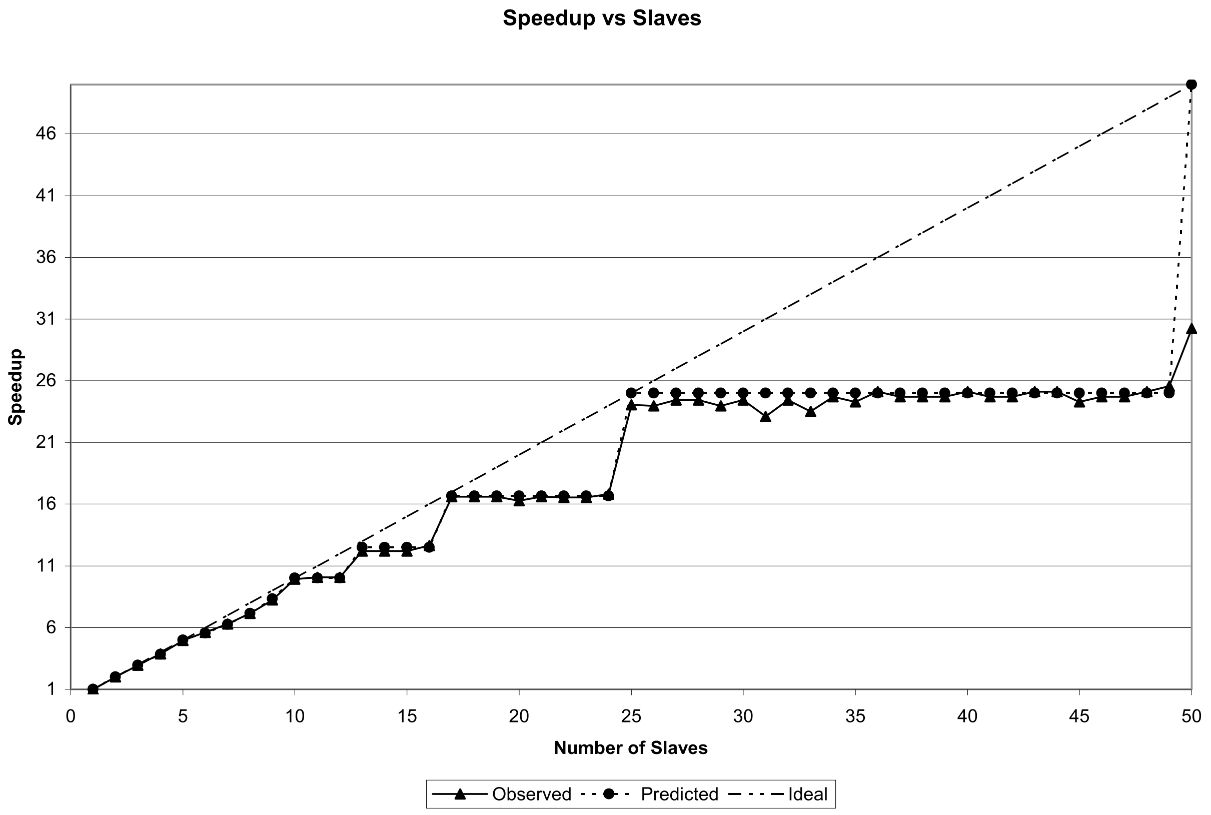

FIGURE 7: Speedup of distributed GAID using Galapagos.

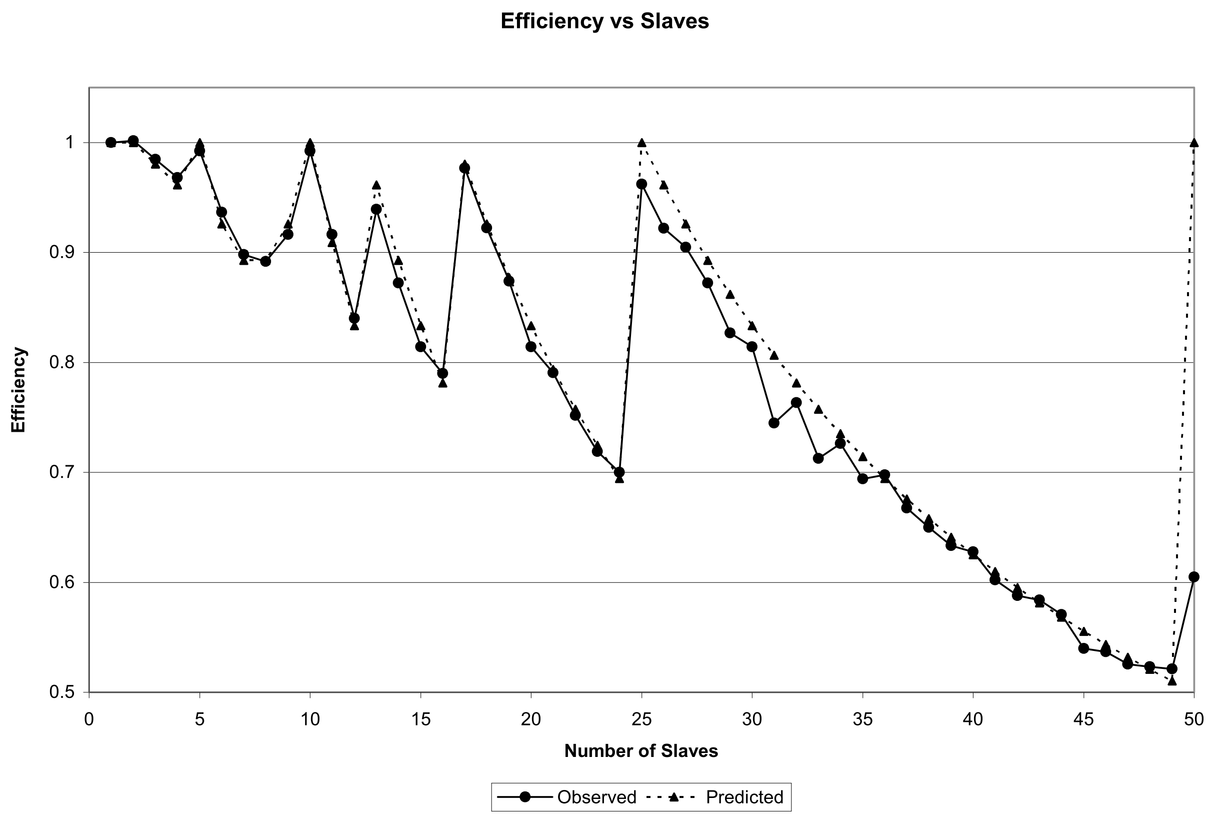

FIGURE 8: Efficiency of distributed GAID using Galapagos.