Galapagos:

a Distributed Parallel Evolutionary Algorithm Development Platform

By
Nicolas Jeremie Kruchten

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE

DIVISION OF ENGINEERING SCIENCE
FACULTY OF APPLIED SCIENCE AND ENGINEERING
UNIVERSITY OF TORONTO

Supervisor: B. Abdulhai

December 2003

Abstract

This thesis describes a new platform, Galapagos, for the development of distributed evolutionary algorithms which can provide useful solutions in practical timescales to difficult optimization problems in the transportation engineering and operation fields. Evolutionary algorithms are powerful stochastic approaches to optimization problems which can find solutions where traditional methods cannot but they can be very computationally demanding. Distributed computing is an approach that has been used with great success to cost-effectively speed up the execution of such resource-intensive algorithms by spreading the load across multiple computers. The Galapagos platform combines these two techniques in a flexible, powerful way so as to enable transportation researchers and practitioners to quickly and easily develop and deploy efficient solutions to previously intractable problems so as to open up new avenues of research in this important field.

Acknowledgements

I would like to acknowledge and thank the following people for their support and help during the months of work leading up to this thesis:

- Professor Baher Abdulhai – for his time, calm support and guidance and his willingness to provide an ambitious undergraduate with the opportunity and resources to undertake such fascinating and rewarding work
- David de Koning – for his donation of the code which formed the backbone of what would eventually become LightGrid and Galapagos
- Lina Kattan – for her patience, support, her much appreciated help in debugging and her never-ending quest for the addition of new and useful features to Galapagos

Table of Contents

List of Figures

1 Introduction

Numerous problems in transportation engineering and other fields can be formulated as optimization problems, albeit ones that are difficult and slow to solve. This thesis describes a new software development platform, Galapagos, which can enable researchers and practitioners to easily develop systems that can rapidly find good solutions to such optimization problems.

In long-term transportation planning, models are of critical importance, but their calibration is often very difficult and only really accomplished through intuition. Calibration in this setting implies reducing an error term by manipulation of a variety of parameters, in a word: optimization. Short-term transportation system planning and operation is essentially concerned with finding the best, most efficient, most cost-effective way of delivering transportation services: finding an optimal solution to this problem. Intelligent Transportation Systems (ITS) applications often rely on a mix of various Artificial Intelligence (AI) approaches and simulations of transportation systems, both of which also heavily rely on solving optimization problems, both in calibration and in machine learning systems.

Over the past 30 years, a class of stochastic algorithms known as evolutionary algorithms (EAs) has been increasingly used both as AI systems in their own right or as optimization tools for AI and in other fields, such as transportation engineering and operations research. This class of algorithm has been found to provide better and faster solutions for optimization problems than traditional, deterministic optimization approaches. EAs essentially take a cue from nature and evolution to ‘evolve’ better and better solutions to a given problem. Much like natural evolution, however, depending on the nature of the problem at hand, an EA’s artificial evolutionary process can take a very large number of generations (iterations) in order to provide good solutions.

Though microprocessors have been becoming simultaneously more powerful and cheaper since the days when EAs were first developed, the high-performance computing field has been looking to harness the power of multiple processors working together rather than one single, extremely fast processor. This parallel computing approach is often simply

more cost-effective, fault-tolerant and practical. Early forays into parallel computing focused on multi-processor computers, but when the price for commodity hardware fell as its power rose, it became cost-effective and easy to put together what are known as clusters of dedicated, cheap computers instead of buying expensive multi-processor machines. In the age of the Internet and high connectivity, the way of the future increasingly looks like grid, or distributed, computing: the idea of assembling single- and multi-processor computers as well as clusters into a sort of fluid 'computation-on-demand' labour pool. A client would simply hand a computational task to the 'grid' for processing, without having to own or maintain any hardware.

There are a variety of approaches to combining the power of EAs with the speed of parallel computing in order to obtain practical systems, but this is still a young field and there is a lack of flexible, extensible software tools to for both research and production systems. Galapagos is a step towards filling this need.

Galapagos is a software development platform: a set of components which form a base onto which developers can build a variety of EAs without having to rebuild the basic EA functionality for every application. Galapagos is built for flexibility, extensibility and reusability so that developers can focus on their application rather than building and testing software which all EAs rely on. Galapagos allows developers to use a broad range of EA types and to run their application in a grid-like setting, taking advantage of the hardware at their disposal.

These grid-computing features of Galapagos are provided by another set of software components called LightGrid. LightGrid is a generalized light-weight grid-computing engine which was developed as a part of Galapagos but can also be used as a base onto which other distributed applications can be built.

This thesis begins with an introduction to EAs, including parallel EAs, in order to describe the functionality which Galapagos must provide to be considered a flexible EA development platform. The next chapter describes distributed computing and defines some terms, metrics and design challenges. Distributed EAs are then discussed, in order again to define what Galapagos must facilitate to be useful in developing distributed EAs. Next, LightGrid and Galapagos itself are described in detail. This work finishes with a

case study of Galapagos in action on an ITS problem of some importance: automated incident detection. This case study serves to showcase Galapagos' various features and to demonstrate the benefits of using Galapagos in developing EA-based solutions to optimization problems.

Appendix A contains step by step instructions to running Galapagos in its default mode once an application is built.

2 Evolutionary Algorithms

This chapter is an introduction to evolutionary algorithms (EAs) so as to define the algorithms which Galapagos is meant to implement. After describing the general form of EAs and defining some terms, the relationship between EAs and optimization is examined. The next section describes the three independently developed 'mainstream' EA approaches and demonstrates how they each fit neatly into the general EA form. Parallel EAs are then described, followed by a section on EA design challenges.

2.1 General Form

Broadly speaking, EAs search some multi-dimensional space for points according to some criteria [1]. They do this by operating on some set of points, applying stochastic operators on them to generate new points, then using some selection mechanism to discard unpromising points, bettering the working set. More formally, an evolutionary algorithm is one that can be described by the following pseudo-code:

initialize P_0
evaluate P_0
let $t = 0$
do
    generate P_t' *from* P_t
    evaluate P_t'
    assemble P_{t+1} *from* P_t *and* P_t'
    let $t = t+1$
until (some stop condition is met)

P_n in this context is the working set of feasible points at the n^{th} iteration, or, in EA terminology: the *population* of *chromosomes* at the n^{th} *generation*. A chromosome is one feasible point, and it is comprised of *genes*: variables which locate the point in the search space. The word generation can interchangeably refer to one particular iteration of the loop or to the population in memory during that iteration. P_n' is the set of *children* of the n^{th} generation.

The evaluation step is where the search criteria come into play. Each chromosome is evaluated and scored according to some scheme: in EA terminology, they are assigned a

fitness. This fitness is then used in the selection process to favour promising points and discard unpromising ones.

This selection process usually comes into play at the assembly step (which is why that step is often itself called 'selection'). During the assembly step, some subset of the current population and its children is selected to become the next population. An EA could potentially apply the selection pressures during the generation of the children, rather than, or as well as, the assembly step.

The children are generated from the population mainly through *recombination* or *mutation* operators though some EAs (notably Evolutionary Programming, covered later in this chapter) have been implemented without recombination. Recombination is an operator whose inputs are 2 or more chromosomes from the population and whose outputs are 1 or more children whose location in the search space was derived from its parents' location according to some (usually stochastic) rule. Mutation is an operator whose input is a chromosome and whose output is a slightly modified version of that chromosome: its location in the search space has been shifted according to some rule.

The use of this general form to describe evolutionary algorithms clearly shows the inspiration that this approach takes from the process of natural selection and evolution which has been famously conjectured to explain the biodiversity of our world. In a reverse analogy then, the chromosomes of the natural world's EA would be strands of DNA (the solution space being the set of all possible DNA strands) and the generation and assembly steps are the processes we see all around us: the formation and continuance of families and the competition for limited resources.

Multiple algorithms that fall into the generalized definition above were independently developed by computer scientists and engineers over the span of a decade from the mid-'60s to the mid 1970's. Some unsuccessful experiments in machine learning in the 1950's could also fall into the EA class [1]. The three 'mainstream' branches of EAs will be covered later in this chapter: Evolutionary Programming (EP), Evolutionary Strategies (ES) and the well-known Genetic Algorithms (GAs).

2.2 Evolutionary Algorithms and Optimization

EAs have been used in a variety of applications, from machine learning to automatic programming to the design of physical objects, and their usefulness depends on the space being searched and the criteria being used to search it. One of the most easily grasped and powerful applications of EAs is optimization: the search for points which maximize or minimize some multi-dimensional function, the *objective function*. Optimization is a fundamental application of mathematics and has almost infinite applications in transportation engineering, and operations research as well as other engineering and scientific disciplines.

Traditionally, optimization techniques have been deterministic approaches: hill-climbing, the simplex algorithm, the Franke-Wolfe algorithm etc. though other stochastic approaches than EAs exist, such as simulated annealing. Furthermore, these techniques have been found to be very limited or problematic when applied to certain problems: their application usually result in the discovery of local, rather than global, optima, if they are applicable at all. A local optimum is a point which is optimizes a function with respect to points neighbouring it, while a global optimum is a point which optimizes the function with respect to the whole search space under consideration.

Traditional methods can be difficult to apply to non-linear or highly multi-dimensional problems in that these objective functions can be difficult or impossible to differentiate. Traditional deterministic search methods can also be very sensitive to their starting point, which can be a problem with very large search spaces. EAs essentially sidestep all of these problems, as they do not require the differentiation of the objective function, merely its repeated evaluation, and they operate stochastically on multiple search points in parallel [1].

EAs are not without their own set of challenges, however. The major drawback to using EAs is that can often require thousands or hundreds of thousands of fitness evaluations before producing worthwhile results. With the computing power easily accessible to many people and institutions today, this may not seem like a major problem, but if each evaluation takes minutes or hours, such EAs can quickly become impractical. Techniques

to mitigate these problems will be covered in a later chapter, after an introduction to parallel and distributed computing.

In order to effectively use EAs in any specific setting, the choice of relationship between the representation of the search space and objective function and the representation of the chromosomes and fitness is critical. In optimization, the chromosome's genes are typically some encoding of the coordinates of a point's location in the search space and its fitness is the value of the objective function at that point, sometimes scaled according to some scheme.

Beyond optimization, other uses do exist for EAs, notably in control systems and machine learning. In these contexts, the search criteria may not be static, but changing over time. The chromosomes may also represent something fundamentally different from coordinates on some axes: they could be behavioural rules, for example.

2.3 History of Evolutionary Algorithms

Evolutionary Programming (EP) was described in the literature by Fogel in 1962 [3] and his book, published in 1966, [3] is considered the landmark work in EP. EP was initially used as an AI technique for the prediction of states in complex systems. In the late 1980's, Fogel's son, still working in isolation from the other branches of EA research, extended EP techniques to such applications as optimization. Evolutionary Strategies (ES) were developed by engineering students in the late 1960's as optimization techniques for the design of flash nozzles. One of these students received his Ph.D. in 1970, [4] for his work in ES (the first published record of ES). The ES field progressed in relative obscurity for 30 years, much like EP, and in fact the two approaches display surprising similarities. Genetic Algorithms (GAs) are the most well-known members of the EA family. Independently developed by Holland and published about in 1975, [5] GAs were first explored as adaptive AI systems than as function optimization tools. The first contact between these isolated EP, ES and GA communities occurred in the early 1990's and slowly the three approaches have been converging towards what are now known as EAs.

There are several interesting and important differences between the three approaches toward EAs, though a thorough comparison is beyond the scope of this work. These differences can largely fall into the following categories: representation, mutation, recombination and selection.

2.4 Representation

GAs as initially developed operated exclusively with strings of bits (binary digits) as chromosomes. An encoding between the problem domain at hand and these bit-strings was required. This approach follows naturally from GAs' initial application as AI tools: they were not initially developed as function optimization tools.

EP was similarly not developed for this application, but rather for state-prediction. Most EP work does not require a specific coding of the problem-domain to a particular format of chromosomes.

ES was initially conceived as real-valued function optimization tools and as such use a 'natural' representation of the problem domain (real number or integers, depending upon the problem.

As GAs developed and were used in new fields and applications, they broadened to include 'real-coded' GAs and other such natural encodings as trees or graphs, evidence of further natural convergence between the three approaches.

2.5 Mutation

Mutation is one of the primary mechanisms whereby EAs explore the solution space. operations are largely linked to the problem-representation. In GAs mutation took the form of random bit-flips (inversion of the bit). In ES and EP mutation took the form of a gaussian stochastic operation: adding to or subtracting from the initial genes in question.

These two general approaches highlight the two major factors in mutation: which genes to mutate and how to mutate them (in early GAs this latter point was moot, bit-flipping was the only option). A large variety of mechanisms have been proposed and studied to answer each of these questions.

One major branch of mutation operators of note is known as *adaptive mutation*. In each of these cases, the scale and form of the mutation changes through the progress of the algorithm run. In the context of continuous real function optimization, this takes such forms as random perturbations of varying radius around the gene in question. This adaptation can depend on any feature of the EA, from the variance of the fitnesses in the current population to what are known in ES (and advanced EP) as *strategy parameters*. These are parameters attached to each gene which change with each generation and are used in determining the size of the potential mutation applied to this gene. The use of strategy parameters gives rise to what is known as *self-adaptation*.

2.6 Recombination

Recombination, like mutation, is a fundamental mechanism or exploration in EAs. It is of note, however, that not all EAs use recombination: standard EP and early ES, for example, focused exclusively on mutation to explore the space. The use of recombination in EAs sheds some light on the importance of using a population of chromosomes to explore a search space. If only mutation is used, it makes little difference whether the population has one or hundreds of chromosomes, as they behave independently from each other.

In recombination, 'genetic information' is taken from two or more parent chromosomes to generate a child chromosome: the child's genes are derived somehow from those of the parents. Recombination, like mutation, depends on the underlying representation of the problem (i.e. real, binary or some other form). Traditional GA recombination took the form of 'crossover' between two parents in which the bit-strings were split at one or more points and the resulting fragments were matched up in various combinations to generate children. ES approaches to recombination included a similar type of operation in which some genes were copied from each parent, but the use of real values as genes also allowed for 'blending' operators which chose values for the children that were somehow between those of the parents. Variants of recombination and crossover operators used more than two parents, and/or generated more than one child at a time.

The mechanism used to choose which chromosomes in the current population to use for recombination can be deterministic, stochastic or can form a part of the selection pressure

by depending upon the fitness of the parents. These mechanisms are further discussed in the section on parallel EAs.

2.7 Selection

The application of selection pressures in an EA usually occurs in the assembly step, and as such, this step itself has traditionally been called 'selection'. In this work, a distinction is drawn between the actual choice of chromosomes for some purpose (i.e. selecting parents for recombination) and the specific action of assembling a set of parents and children into a new population by discarding some of them.

As with mutation and recombination, a wide variety of schemes have been proposed and successfully used as an assembly step. Traditional GAs simply replaced the parents with the children, but later variants had small numbers of children which 'crowded' into their parents' population, displacing some of them. Which chromosomes were displaced was addressed either deterministically or stochastically, and/or depended upon their fitness. ES and EP practitioners independently developed similar approaches toward this fundamental EA step, though it is possible to imagine successful EAs which simply do not reduce the population size, and keep all parents and children throughout their run.

2.8 Parallel Evolutionary Algorithms

An important subset of EAs is known as *parallel EAs* [1]. In this type of EA, the population displays some form of internal structure which comes into play when parents are selected for recombination. In *panmictic* EAs, any two chromosomes in the population can potentially be recombined with each other. In *diffusion-style* parallel EAs, the chromosomes are taken to be somehow spatially distributed (in a two-dimensional grid or higher-dimensional structure) and only neighbouring chromosomes may be recombined. Another model of parallel EAs is the so-called *island model*, in which semi-independent sub-populations, or *demes*, essentially behave as separate EAs, periodically exchanging chromosomes through a process known as *migration*.

This type of EA displays even more similarity to the theory of evolution of species than most. Darwin, before publishing his famous work, traveled to the Galapagos Islands off

the coast of South America, to study finch populations and how they evolved differently on different islands. Diffusion-style EA processes can also be observed in nature when one species is examined at different locations in the same continuous geography.

2.9 The State of the Art

The EA field is still feeling the effects of almost 30 years of independent development of its major branches. GA, ES and EP researchers all developed their own terminology and biases over time and this comes through in any survey of the literature. Cross-pollination between these branches and the various fields in which EAs have been applied has been very beneficial to the EA community as a whole, but fragmentation still exists.

The wide variety of operators and approaches to the same EA issues and steps highlights the tentative and intuitive nature of EA work. Precious few predictive theoretical results exist to guide the EA practitioner in the construction of an application, but some heuristics do exist with respect to the choice of such parameters as population size and strength of selection and mutation pressures [6]. Notably, the use of parallel EAs has been found to be beneficial in terms of convergence times and quality of solution (i.e. local versus global optima) [7].

As a new generation of EA researchers and practitioners emerges, one whose education does not predate the field's convergence, we can expect further consolidation of the field and an increase in the number of general results rather than a continued fragmentation along GA, ES and EP lines. It is hoped that the subject of this work, Galapagos, is one step in this direction.

2.10 Design Challenges

EA designers and users face some substantial challenges when applying EAs to a given problem. The bewildering variety of operators and parameters without strong theoretical analytical tools is a major barrier to entry to this field, as are the difficult choices of representation of chromosomes and fitness. Once past these hurdle and actually using EAs to solve optimizations, however, one major design tradeoff often encountered in the choice of operators and calibration of parameters is that of convergence speed versus

solution quality. In many cases EAs are more likely to converge to ‘good’ solutions to optimization problems than traditional methods, but this can take a long time and often, the quality of the local optima found is difficult to evaluate in the absence of information about any global optima.

The Galapagos platform provides a flexible implementation of the EA pseudo-code described in this chapter in order to allow EA developers to focus on EA-related design problems rather than on the mechanics of implementing the EA basics. Each step of the EA pseudo-code is completely developer-definable and Galapagos thus allows for the study and use of any of the EA types discussed in this chapter. Furthermore, Galapagos was built in such a way as to encourage sharing and reuse of components such as EA operators, so as to further the EA field in general and speed up development times.

3 Distributed Computing

This chapter is an introduction to the current state of parallel computing and distributed computing in order to provide a basis for the discussion of distributed EAs in the next chapter. After defining some performance metrics for distributed computing, design challenges in this field are outlined. The discussion of grid computing also serves to illustrate the utility of the LightGrid engine of which the Galapagos platform itself is an application.

3.1 Parallel Computers, Clusters and Grids

For the past 25 years, the semiconductor industry has been matching the predictions of Moore's Law: doubling the number of transistors per area on a microchip every 18 months. This has led to a drastic drop in the price of computers as well as simultaneous rise in their power. Paradoxically, the cost-power ratio for single processors at any given time is often highly non-linear: it can cost much more than twice the price of a slow processor for one twice as fast.

Due this manufacturing and economic reality, the high-performance computing field has for years been looking towards systems with large numbers of processors working together to solve problems as more cost-effective solutions. These types of systems are also often more fault-tolerant than single-processor systems; if one node breaks down, the rest can still function. There are three major paradigms that multi-processor systems can fall under: parallel computers, clusters (also known as Beowulf clusters) and grids.

Parallel computers are single computers with multiple microprocessors. Some commercially available multi-processor systems exist, notably in the dual-processor variety, but the world's extremely powerful parallel supercomputers are usually one-of-a-kind purpose-built devices. This kind of computing was the earliest form of commonly-used parallel computing, and software that runs on these machines has to be written or compiled so as to make effective use of the multiple processors.

With the rise of low-cost, mid-power consumer computers came the concept of the Beowulf *cluster*. This term has no precise definition but as a general rule such clusters are

sets of identical off-the-shelf consumer-grade computers which are dedicated to the purpose of being a part of the cluster. They most often have parallel computing software installed, such as the Message Passing Interface (MPI) or the Parallel Virtual Machine (PVM) that enable them to function as if they were a single parallel computer (there is no software package called 'Beowulf'). Clusters are basically a way of attaining computational speeds similar to small supercomputers, but at a much lower cost. The use of off-the-shelf consumer components makes them very attractive for companies and institutions without the resources to order or maintain a purpose-built machine.

The concepts of *grids* came about with the rise of widely available and reliable Internet connectivity. A grid is a pool of often non-dedicated computers and clusters that offer services such as the ability to store data or perform discrete computational tasks known as work units or *jobs* [8]. The idea behind grids is that anyone with 'spare cycles' or unused computer resources could donate or sell its usage to the grid, and that some sort of accounting or dispatching process exists to coordinate the system.

Grid computing (also known as distributed computing, because the computers are often geographically dispersed) has been in use in certain fields and applications for some time, but there is not yet one Grid to which anyone can conveniently hand computational tasks to. Notable applications of grids have been to allow the general Internet population to download 'screensavers' that provide interesting graphics when their computer is idle, while it computes work units such as Fast Fourier Transforms for the Search for Extraterrestrial Intelligence (SETI) or molecule-geometry calculations for pharmaceutical research such as the cure for cancer or AIDS [9, 10]. In these cases the grid is not general-purpose, as the grid components always perform the same kind of jobs, but the SETI@Home platform has recently made moves to open up their system, bringing it closer to a generalized grid structure.

3.2 Performance Terms & Measures

Regardless whether the approach towards the parallelization of computing tasks is a parallel computer, a cluster or some other form of distributed computing such as a grid, certain terms and measures have been developed to compare and discuss performance issues.

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

- *Speedup*: the ratio of execution speed of n processors versus 1 processor. (The ideal result in parallel 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 and efficiency change 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.

Speedup is the major performance metric when examining the suitability of a computational task for parallelization. It depends largely on the ratio between the portion of the problem which is serial and the portion which is parallelizable. In any algorithm or computation, there are some parts which are executed sequentially but do not depend upon the results of the previous part. These can be executed in parallel if there are processors available to do so. There are also parts which cannot be executed in parallel. The relationship between speedup, the number of nodes, the serial fraction and parallel fraction can be expressed with the following, which is known and Amdahl's Law [11]:

$$S(n) = \frac{Ts + Tp}{Ts + \frac{Tp}{n}}$$

Where Ts is the serial fraction, Tp is the parallel fraction and n is the number of nodes. Clearly, the larger the parallel fraction, the closer to linear speedup the system will run. 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 (added serial costs when a task is parallelized) but some have suggested various justifications for the existence of super-linear speedup, including hidden costs in serial computing such as limited memory and other issues [12]. 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.

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 scalability and adaptability 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.

There are variations and extensions to Amdahl's Law which show the limits of system scalability (i.e. they put an upper bound on the number of processors which increase speedup) but these are out of the scope of this thesis. Robustness and adaptability are largely qualitative measures.

3.3 Design Challenges

High-performance computing is a multi-disciplinary field fraught with physical, economic and electronic and software engineering challenges. Semiconductor manufacturers encounter physical limits as they make increasingly smaller and smaller transistors. Their ability to market various microchips and the high cost of retooling chip fabrication processes places economic limits on what kinds of microprocessors are manufactured. Electronic and software engineers must work together to design hardware that is flexible and software that will efficiently execute on this hardware.

One key concept when dealing with distributed or parallel computing is that of *granularity*, or the ratio of computation time to communication time for a given unit of work performed by a node in a parallel system. Granularity is a function of the problem at hand, the power of the node and the speed of the interconnection between nodes (the latter is quantified by the time per communication, known as *latency*). The design of parallel or distributed applications requires an understanding of these issues in order to produce efficient systems.

When dealing with grid computing or other applications in which the hardware working on the problem is not necessarily under the direct control of the user, security and data

validity issues come to the fore: one must be able to trust that the job results being returned by the grid are correct and not a malicious attempt to interfere with the system.

Galapagos enables EA developers to build EAs as distributed applications (which is discussed in the next chapter) and this is made possible by its underlying grid-computing engine: LightGrid. LightGrid is a light-weight, low-security system which simply comprises of a dispatcher component and templates from which one can build clients and define jobs to be performed. Galapagos does just that, as is described in the next chapter.

4 Distributed Evolutionary Algorithms

This chapter describes some architectures for using distributed computing to speed up EAs. They are then combined with a grid-computing approach towards distribution, in order to explain how Galapagos and LightGrid enable developers to assemble distributed EA applications. The chapter concludes with a description of the challenges distributed EA designers face and how Galapagos can let them focus on these rather than the mechanics of the programming for these applications.

As a note to the reader: there is an important distinction between the 'parallel' in parallel EAs and the 'parallel' in parallel computing. Parallel computing distributes the computation over multiple processors, all executing simultaneously (i.e. in parallel); whereas, the 'parallel' in a parallel EA usually refers to the structure of the population. A parallel EA can execute sequentially on a single processor, and the parallelism in the evolution of the demes is simulated. By convention in this thesis, the term 'distributed' instead of 'parallel' is used when referring to computation, and use 'parallel' only when referring to the EA's population structure.

4.1 Appropriateness of Distributing Evolutionary Algorithms

Having established that EAs are powerful yet potentially slow solutions to important problems and that distributed or parallel computing can be a cost-effective solution to such speed issues, the natural question is: can we combine the two effectively? The answer is a resounding yes.

EAs are of a class of problems (in high-performance computing, the algorithms are the problems) known as 'embarrassingly parallel'. In the discussion of parallel computing performance metrics in the previous chapter, the idea of a serial and parallel fraction of an algorithm was introduced. The larger the ratio of parallel to serial portions of an algorithm, the closer we can expect to get to linear speedup. As mentioned in the chapter on EAs, the most time-consuming step is most often the fitness evaluation step, which can take minutes per evaluation, and require hundreds of evaluations per generation. In a given generation, however, the evaluations do not depend upon one another, making the

evaluation step of an EA easily parallelizable and making the serial fraction (the EA operators and population management) seem almost negligible in comparison.

In short, we can expect to regularly achieve close to linear speedup when we distribute EAs. In fact, some researchers even claim to have observed super-linear effects [13], though this result can be considered controversial, given the issues with the very existence of super-linear speedup.

4.2 Master/Worker Architecture

One of the simplest and most obvious ways to distribute an EA's computation is what is known as a master/worker architecture. In this context, one processor takes on the role of the master, managing the population and the genetic operators and 'farming out' the task of evaluation to the other processors. In this way, the evaluation step is parallelized directly. The following diagram illustrates the principle.

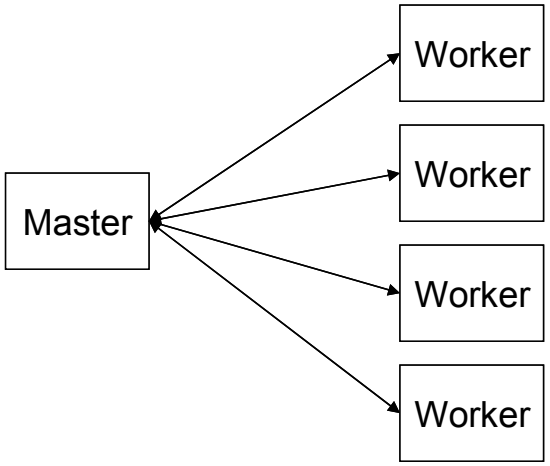

Figure 4-1 Master/Worker Architecture

This sort of architecture is appropriate in situations where each processor has easy, rapid access to the others and the fitness evaluation costs much more (i.e. takes more time) than the communication latency. This is rather fine-grained application: communication between the processes is frequent compared to the peered architecture presented below.

4.3 Peered Architecture

When dealing with island-model parallel EAs, peered architecture is a natural fit in terms of distribution. Under this scheme, each processor in the system manages one population and performs the evaluations proper to its subset of chromosomes. Communication is infrequent, as it only occurs during migration from one deme to another. For this reason, island model parallel EAs are also known as coarse-grained parallel EAs (in this context, the granularity refers to the population/migration structure) and in contrast, diffusion-style parallel EAs are known as fine-grained.

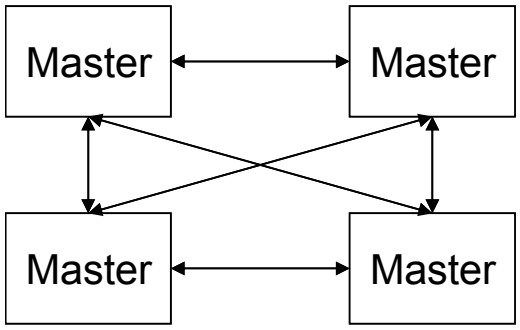

Figure 4-2 Peered Architecture

This architecture is appropriate for situations where one is implementing an island-model parallel EA and the latency is known to be high compared to the cost of an evaluation (i.e. if the cost of one communication is 10 times that of an evaluation, it makes a lot of sense to perform many evaluations on one processor and to communicate infrequently). In some cases, it is conceivable that the EA operators themselves are quite costly and it might be necessary to parallelize this load across multiple processors in this way.

4.4 Hybrid Architectures

The two EA distribution approaches presented above can be combined in the following way: each peer in the peered architecture could in fact be a master, farming out evaluations to its own pool of workers.

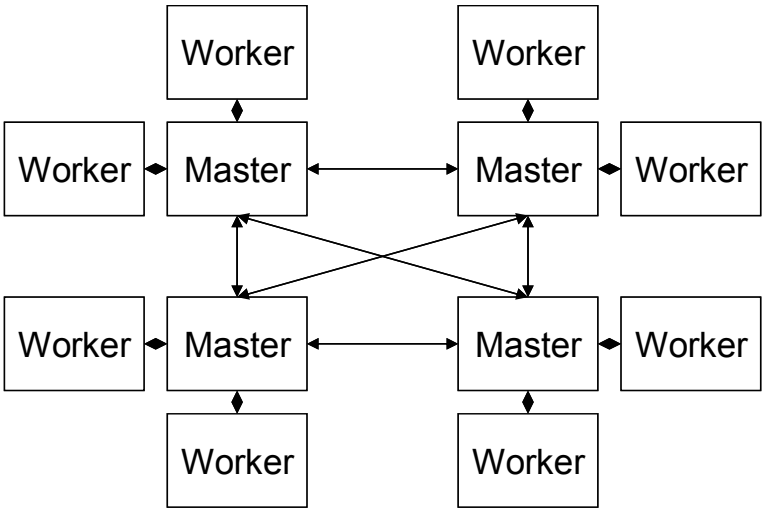

Figure 4-3 Hybrid Architecture with dedicated workers

Taking this idea one step farther, one can envision a situation in which the masters share a pool of workers, as illustrated in the following diagram (in a slightly rearranged form):

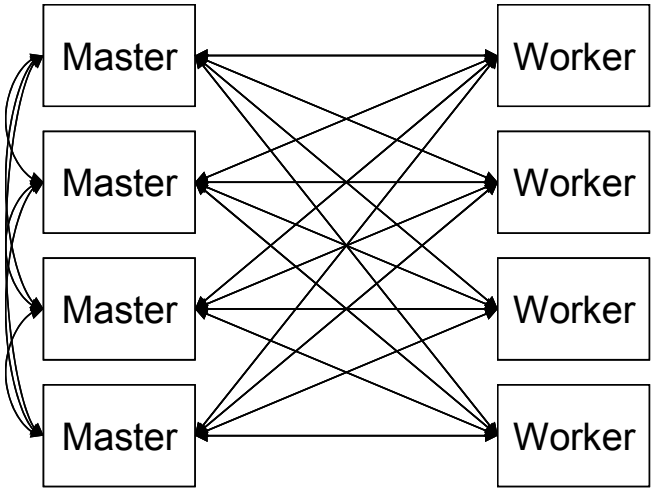

Figure 4-4 Hybrid Architecture with shared workers

The above situation strongly resembles the canonical setup for grid computing, in which a set of clients can hand jobs to a grid, comprised of a pool of available worker processes, via a dispatcher process and this is exactly the model Galapagos and LightGrid provide in order to enable developers to build distributed parallel and/or panmictic EAs:

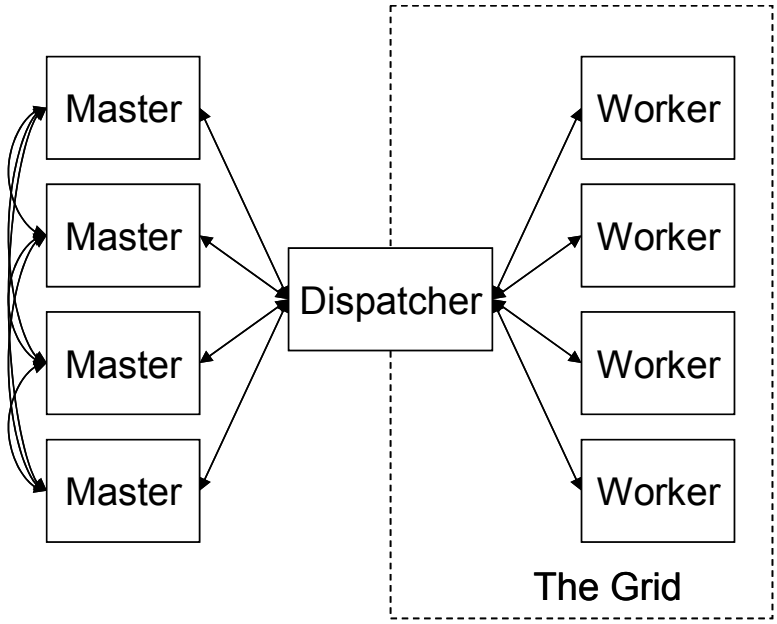

Figure 4-5 Hybrid Architecture with shared workers in a grid with a dispatcher

4.5 Design Challenges

If the design of EAs and of distributed applications are each difficult in their own right, the problem of designing efficient, integrated distributed EAs only serves to amplify the difficulties. The choice of population structure, operators and representation all affect the size of the parallel and serial fraction, the granularity of the application and the type of distribution scheme that is possible or appropriate. On the other hand, the available computer and network hardware can also influence the design of the EA.

Starting in the next chapter, the way in which LightGrid and Galapagos can enable developers to easily build such applications will be explained, as well as how by using them, the developer can focus on the design challenges inherent in this field, rather than the mechanics of programming and debugging a complex application.

5 The Galapagos Platform

This chapter presents the subject of this thesis: the Galapagos platform, a distributed parallel evolutionary algorithm development tool. Galapagos is itself built on top of another engine: LightGrid and thus this chapter first discusses the design philosophy underlying both pieces of software, and then LightGrid and Galapagos are described in detail.

5.1 Background

Galapagos grew out of a project whose aim to create a distributed version of a specific GA application for ITS: an automated traffic incident detection system based around an artificial neural network calibration problem. This application is known as GAID. After achieving the goal of this precursor project, the need for a more general and widely applicable version of the same code became obvious as there are many other ITS problems that could be tackled by computationally intensive GAs. Later, the focus of the platform's development was again broadened to encompass EAs as a whole, and not just GAs.

Early on in the development of a generalized version of the distributed GAID application, it became clear that the EA component and the distribution components had relatively little required coupling in terms of design. The code required to manage the dispatching of jobs and results from one computer to another and back is fundamentally unrelated to such computational tasks as the actual fitness evaluation which occurs on one end of this chain and the EA operators which govern the other end. As such, the code, which was modular to begin with, was split into two components along these lines. One became LightGrid and the other Galapagos.

5.2 Design Philosophy

During the development of Galapagos, a serious attempt was made (and largely successfully achieved) to follow software development best practices. The whole platform was built in a programming language called Java, which is an object-oriented language. In a nutshell, this means that all of the components of Galapagos are self-

contained units of program code and associated data structures. The advantages of object-oriented programming in software engineering have been well-documented [14] and among them are the ease of encapsulation of data and code and the ease of extending these modular components. Both of these features were heavily used in the development of Galapagos and this will be highlighted in the following sections.

The whole design philosophy behind the platform is one of generality, openness, reusability and extendibility. Wherever possible, design choices were made so as to leave the developer with as much freedom in his or her own design as possible. Throughout the design and implementation process, the code and architecture was regularly examined in order to seek out the underlying assumptions made about possible uses for the platform. Wherever possible, the design was modified so as to challenge these assumptions and leave the platform more useful to more people as a result.

5.3 LightGrid

LightGrid is the light-weight grid-computing engine which was 'spun out' into a separate platform during the implementation of Galapagos. It consists of a job dispatcher and worker-type resource component as well as code templates (Java abstract classes) which can be extended to form complete distributed computing applications.

5.3.1 Component & Functional Description

LightGrid is not a complete application, but contains two complete components and two templates which can be extended to create a complete application. The dispatcher and resource components are complete and self-contained and the job-processor and client components are templates which are meant to be extended.

A LightGrid client is a process that can send 'jobs' to the LightGrid dispatcher. These jobs consist of some data and a tag indicating what sort of processing needs to be performed with it. A LightGrid resource is a program which runs on a machine whose operator has allowed it to become a part of the grid. The dispatcher manages the grid, receiving jobs from clients, dispatching them to available resources, retrieving the results from these resources and passing them back to the client. The dispatcher thus provides a

layer of abstraction between the client application and the worker implemented by the resource.

LightGrid is built to be fault-tolerant and to manage a fluid pool of resources. If a resource crashes, slows down or is taken off the grid by its owner or operator, the dispatcher will reassign the job as efficiently as possible so as not to disrupt the progress of the client application. The dispatcher can manage a very large number of heterogeneous clients and resources simultaneously, and can thus be the hub for a variety of applications at the same time.

In order to develop a new LightGrid application, the developer must first build his or her application as appropriate, using the LightGrid client component as a template or a sub-component of a larger application. The developer must also create a job-processor component, defining the processing which must occur on the resource processor. At run-time, then, the client will send jobs to the dispatcher, specifying the particular job-processor is to be used with the data that comprises the job request. In order for this system to work, each resource must have the job-processor component locally available, an indication that the operator of the hardware is willing to let his or her resources be used for this task.

5.3.2 Comparison to Existing Tools

LightGrid is a 'light-weight' grid engine, in the sense that it is not meant for deployment in public applications. LightGrid provides no guarantees as to the correctness of the results provided, as it relies upon trust between the operator of the application and the operators of the resources. It is recommended that LightGrid only be used in relatively controlled environments such as university or company computer labs or server farms.

Other grid-computing engines and platforms are available commercially and as open-source applications, such as the Globus Toolkit and the Sun Grid Engine, and LightGrid is not meant as competition to these 'heavy-weights'. These systems offer certain security guarantees to their users, as well as a variety of grid-services beyond simple job-dispatching.

LightGrid is appropriate for academic research and production-grade applications known to run in controlled, safe or secure environments such as settings where all of the hardware and network connections are controlled by the person or people running the application.

5.4 Galapagos

Galapagos is the primary subject of this thesis work. It is a platform onto which specific EA applications can be built: it is an implementation of the generalized EA pseudo-code described in the second chapter of this work and defines each step, or operator, as a template which can be extended so as to build a basically limitless variety of new EA operators. Furthermore, Galapagos at this time contains a substantial number of such operators, which can be assembled into a variety of functional EA types.

5.4.1 Galapagos Components

Galapagos consists of two major components: containers and a controller. In its current incarnation, it is built as an application on top of LightGrid and as such makes use of the latter's resource and dispatcher components as well. The Galapagos container is in fact a LightGrid client and the Galapagos fitness sub-component is a LightGrid job-processor module, as shown in the following table:

Galapagos Component	Corresponding LightGrid Component
Container	Client
Evaluator	Resource
Fitness	JobProcessor
Controller	n/a
n/a	Dispatcher

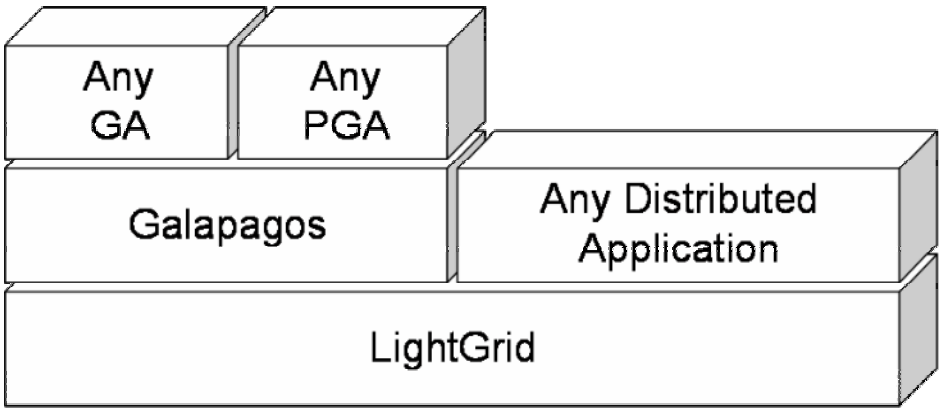

Figure 5-1 LightGrid and Galapagos layers

All of these software modules can be combined together to implement the generalized distributed EA architecture described in the previous chapter, as follows:

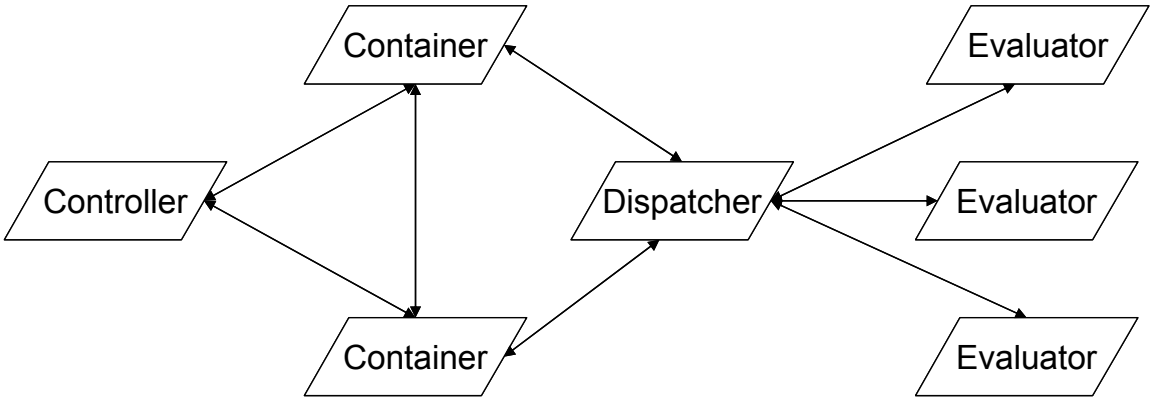

Figure 5-2 Generalized Galapagos process diagram

In the diagram above, the container components take the place of the distributed EA 'masters', managing the EA's demes, or, in the case of a panmictic EA, the whole population in a single container. It is also possible to manage multiple demes in a single container, in order to run Galapagos with the master/worker architecture described in the previous chapter. The LightGrid resources take the place of the 'workers' from the previous chapter, receiving chromosomes and evaluating their fitness as LightGrid jobs.

The Galapagos controller module mentioned above was not present in any of the previous discussions about the distribution of EA computational load. This component is the one which issues all of the control commands for a Galapagos EA run. The controller is a

necessary component as it is where a current copy of the global population (i.e. a copy of each deme) is kept and thus only it can appropriately assess whether the convergence criterion has been met. The controller is also the component with which the user interacts and which does any data logging or output for the EA run.

The Galapagos controller component can easily be integrated in a larger application, and can in fact encapsulate an entire EA run as one function call. A developer could therefore develop an application which simply treats the entire Galapagos platform as a single function, by integrating into it a controller module and using that module's output without the interaction of a user required between the larger application and Galapagos.

5.4.2 Definition of the Evolutionary Algorithm

The first step to building an EA is to define the chromosome representation and the fitness function. Galapagos can use any data structure to represent chromosomes, leaving this choice up to the developer: a template for defining chromosome representations is provided and a real-valued chromosome component already exists. The EA's fitness function is also defined by a template (itself derived from the LightGrid job-processor template) which takes a chromosome as an input and produces a real value as an output. The mechanics of how the fitness is computed is defined by the developer.

In many ITS applications, the objective function will depend upon the output of some other program (i.e. a modeling or simulation package) and in this case, the fitness component will act as a 'wrapper', itself calling the external program, gathering and interpreting its output and converting it into an EA-friendly fitness value.

Galapagos also contains templates for components which represent each of the EA steps, namely: Initializer, Generator, Assembler and Convergence. These templates define the inputs and the outputs of each of these code modules, and anyone can sit down and create a new operator that internally performs whatever function desired.

As discussed in the chapter on EAs, the generation step usually comprises of a mutation operator which operates on the output of a recombination operator, and the assembly step usually contains some sort of selection process. Furthermore, recombination also requires some sort of selection mechanism to determine which members of the current population

to recombine. Galapagos contains templates for each of these three operators as well: Recombiner, Selector, and Mutator.

Operators which extend the Mutator template are intended to be used in conjunction with those that extend one other operator template: GeneMutator. In this scheme, the Mutator component contains decision rules for which genes are to be mutated, while the GeneMutator component defines how this mutation is to occur.

The final set of operators that Galapagos provides templates for is comprised of the Epoch, Migrator and Topology operators, which together define the interaction between the various demes in an island-model parallel EA. Diffusion-style parallel EAs can be implemented by using a Selector module that operates according to some internal population structure.

To summarize, Galapagos provides templates for the following EA operator modules:

- Initializer
- Generator
- Assembler
- Convergence
- Recombiner
- Mutator
- GeneMutator
- Selector
- Migrator
- Epoch
- Topology

Beyond simply defining these operators through templates, Galapagos contains a library of actual operators developed using these templates that can be assembled to produce a wide variety of EA types (the operators that are a part of the library at the time of writing are largely applicable to the real-valued chromosome component also present in the Galapagos distribution).

By defining almost all of its components as templates, Galapagos can be extended as the developer sees fit, to any problem representation, any fitness function, and any set of EA operators.

More information about the specifics of configuring Galapagos for an EA run is provided in Appendix A.

5.4.3 Physical Distribution

Beyond providing developers with a highly modular way of defining EAs, Galapagos enables them to define with some degree of precision how the computation is to be distributed. As detailed in the description of Galapagos components, the container takes the place of a distributed EA master and the evaluator that of a worker. Given a set of computer available for a Galapagos run, the developer can choose on which the containers will run and on which evaluators will run. Furthermore, through judicious usage of dispatchers, containers can operate with one common pool of evaluators or each with a dedicated set.

The following diagrams will illustrate some of the possible mappings of Galapagos' component processes to a set of computers (the computers are the grey rounded boxes):

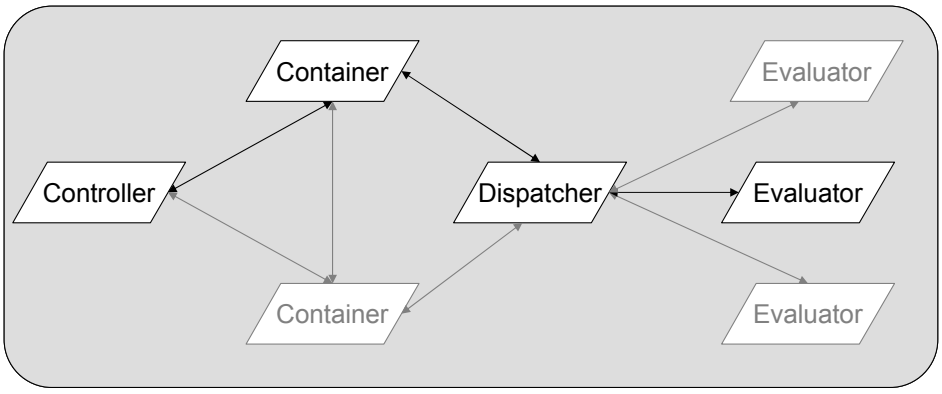

Figure 5-3 Single-computer mapping

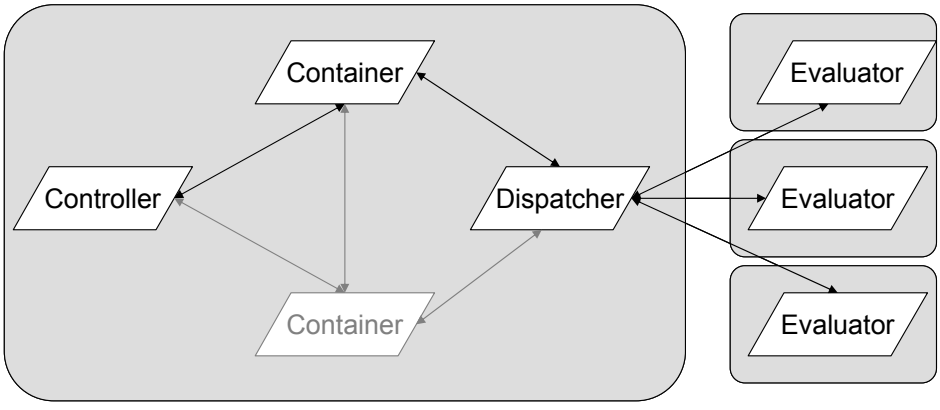

Figure 5-4 Master/Worker Mapping

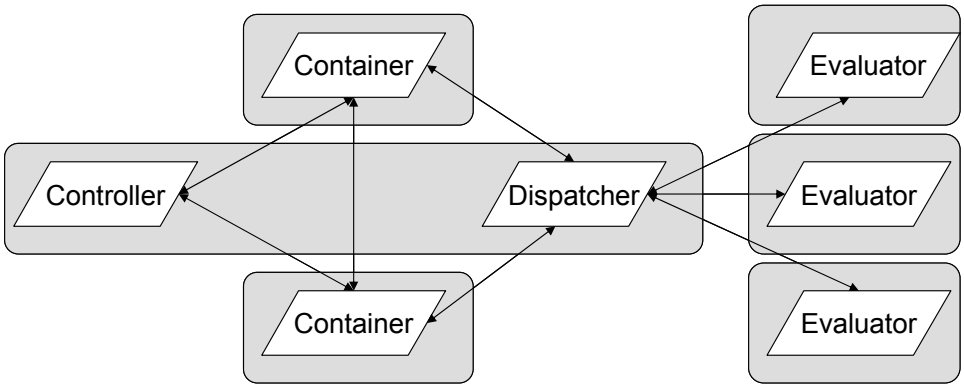

Figure 5-5 Peered Mapping (Controller & Dispatcher collocated)

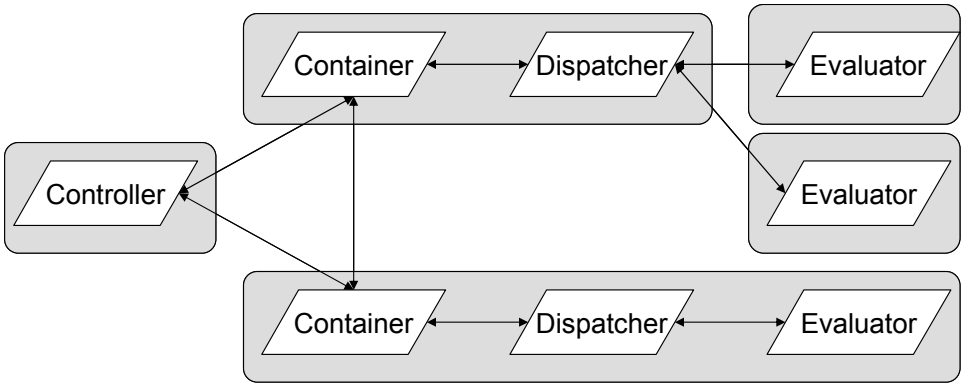

Figure 5-6 Possible highly tailored mapping

Galapagos developers and users can thus tailor the distribution of their EA application to the specific setting in which it is meant to run, taking into account the interaction between the properties of the hardware at hand and the nature of the EA operators and fitness evaluation function.

5.4.4 Comparison to Existing Tools

A variety of GA and EA tools and libraries exist, both commercial and open-source, but few can offer the flexibility of Galapagos in terms of both specification of the EA flow and control over the physical distribution of the computational load. Libraries and specifications exist which define EA and GA operators, and others exist which are meant to allow developers to build EAs which can run on parallel computers (i.e. PGAPack [15]). GA add-ins also exist for standard mathematical tools such as MATLAB. In

general, however, these libraries and tools are not extensible in the same way as Galapagos is, and are not meant to run in a distributed setting.

A project similar in scope and implementation to Galapagos is the open-source DREAM [16] project, which features grid-computing capabilities and tie-ins to some standard EA libraries and specifications. Galapagos' early development focused on GA capabilities and only later converged to encompass features supported by the DREAM project. Galapagos is noticeably simpler and less obfuscated in structure and was independently developed with ITS applications and practitioners in mind.

6 Galapagos Applied to ITS Problems

This chapter contains experimental data from the successful application of Galapagos to actual ITS problems. Galapagos has been rigorously tested, both at the component level and as a whole platform, using unit tests and sample problems. It has also been used to implement two ITS research applications: GAID, the automated incident detection system which the platform grew out of, and a dynamic origin-destination estimation system which is still under development as a part of another student's doctoral work.

6.1.1 Genetic Adaptive Incident Detection

One application of EAs to ITS problems explored at the University of Toronto ITS Centre and Testbed is in the area of automated incident detection. GAs were successfully applied to the problem of training an artificial neural network to differentiate between 'incident' and 'normal' conditions on a stretch of highway. This application is known as GAID, or Genetic Adaptive Incident Detection. The word adaptive is used because the idea behind GAID is to constantly be retraining the classifier using up-to-date observed data. In this manner, the system would be able to adapt to local and seasonal traffic conditions. [17]

GAID's fitness function is formulated as a score reflecting a percentage of correct classifications and false alarms based on a known training data set. The neural network is defined in terms of 16 smoothing parameters. GAID was the first implementation of an EA as a distributed Galapagos application.

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 100 megabit Ethernet network switch where every computer can easily communicate with every other one. The Galapagos evaluators were run as background processes on these publicly accessible machines while normal usage of these computers by engineering students continued undisturbed.

The conclusion of the experiments that were run was that 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 fault-tolerant grid-computing nature of the LightGrid engine also proved to react robustly to large changes in evaluator performance or instances where evaluator machines crashed or stopped returning results.

The expected time to convergence can be estimated with the following reasoning: the distribution of GAID using Galapagos did not modify the structure of the panmictic algorithm, it just distributed the parallelizable step, so GAID on one or many computers should and was observed to 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 evaluators
- g = the number of chromosomes per generation
- $v_i(x)$ = the 'instantaneous computation speed' in terms of 'evaluations per time' of evaluator 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 evaluators 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 evaluator'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 genetic operator steps in the algorithm execute in an almost negligible time compared to the evaluation of a chromosome.

The 50 computers available 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 plotted 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.

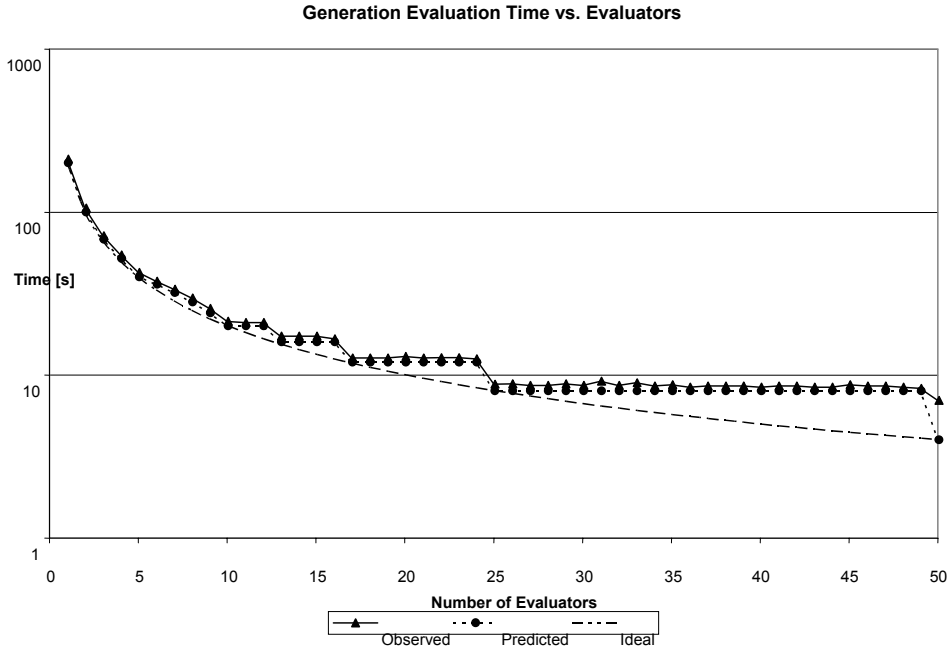

Figure 6-1 Generation Evaluation Time vs. Evaluators

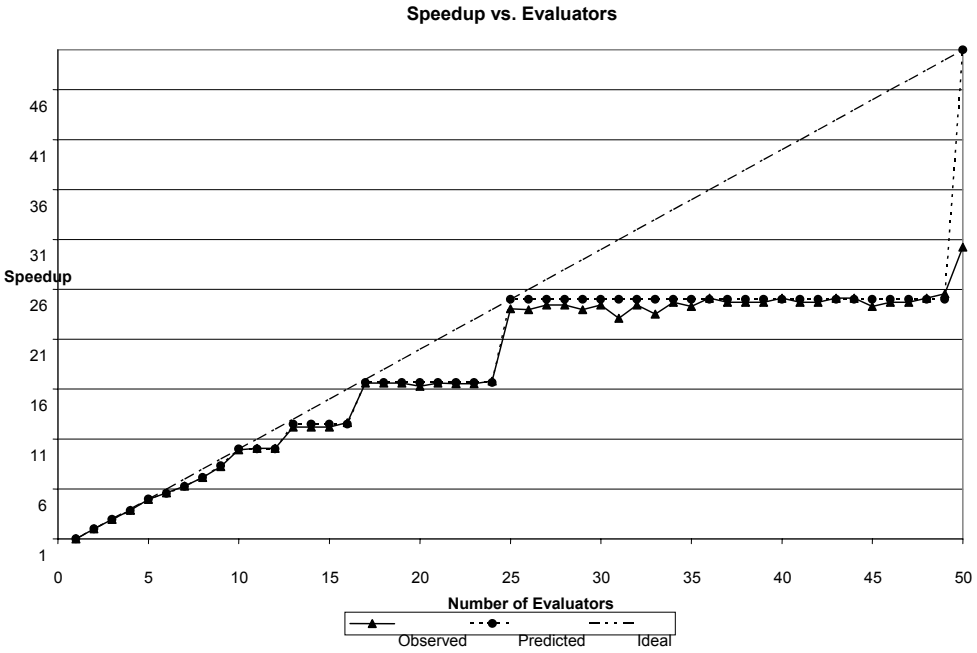

Figure 6-2 Speedup vs. Evaluators

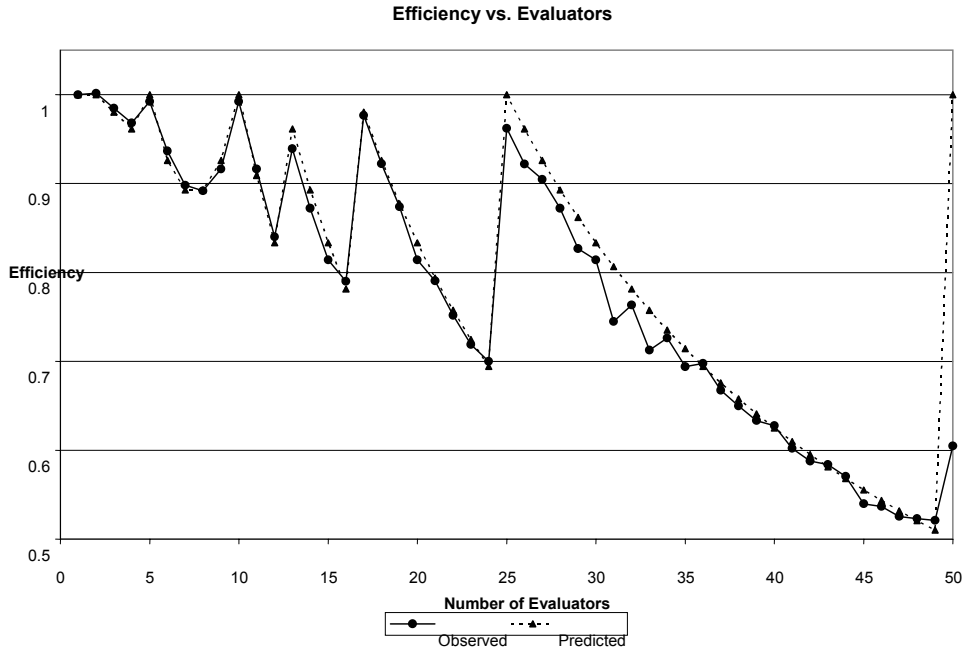

Figure 6-3 Efficiency vs. Evaluators

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.

6.1.2 Dynamic O-D Estimation

Another area of research at the University of Toronto ITS Centre and Testbed is the application of EAs to the problem of dynamic origin-destination estimation. This problem can be formulated as a highly multi-dimensional minimization problem in which the objective function is an error term between some observed traffic data and the output of a dynamic traffic assignment simulation. The EA is to be 'seeded' with a-priori estimates of the O-D matrix such as surveys or the output of previous runs of the application.

This application is still in the experimental stage and as such hard performance numbers are not available. However, it has been successfully implemented using the Galapagos platform. The fitness function is derived from interaction between Galapagos components and the Dynasmart dynamic traffic assignment software, demonstrating concretely that this platform can easily be used in conjunction with existing external software. The computational load for this problem is extremely high, with the EA running on a single computer for up to three days at a time in early trials. Similar results were observed using six computers running for around 12 hours. It is expected that linear speedup will be observed with this application much as they were with GAID and that it will easily scale to the 25 computers available to run Dynasmart (the population size for this application is much greater than, and could easily be a multiple of, 25).

7 Conclusions

This thesis work began with the assumption that ITS problems are often optimization problems and that evolutionary algorithms were powerful but potentially slow solution methods for such problems. Furthermore, it was known that combining the power of distributed computing with EAs could result in substantial gains in speed. The goal of this work was thus to produce a software platform that would enable developers (such as ITS researchers and practitioners) to easily build and deploy distributed EAs.

Galapagos embodies the successful attainment of this goal. It is a very flexible and powerful platform with which one can build a variety of EA applications that can be deployed in a variety of settings. It also enables independent developers to share and reuse each other's EA components, operators and chromosome definitions, as well as design and test new ones. Galapagos also provides the benefit of speeding up development time and allowing developers to focus on the issue of designing efficient and effective distributed EAs rather than the mechanics of programming them. Galapagos has been successfully used to implement two ITS-related applications to date and has proven to bring significant speed gains to both.

Looking to the future, Galapagos can and will be used at the University of Toronto ITS Centre and Testbed to implement rapid, cost-effective solutions to a variety of other ITS and transportation related problems that were previously considered impractical due to the limits of the computational hardware available.

References

[1] T. Back, *Evolutionary Algorithms in Theory and Practice: evolution strategies, evolutionary programming, genetic algorithms*. New York: Oxford University Press, 1996.

[2] L. Fogel, "Autonomous Automata". Industrial Research, 4:14-19, 1962

[3] L. Fogel, A. Owens and M Walsh, *Artificial Intelligence through Simulated Evolution*. New York: Wiley, 1966.

[4] I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Fromann-Holzboog, 1973.

[5] J. Holland, *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press, 1975.

[6] E. Cantú-Paz, *Designing Efficient and Accurate Parallel Genetic Algorithms*, PhD Thesis, 1994

[7] E. Alba, and J. Troya, "An Analysis of Synchronous and Asynchronous Parallel Distributed Genetic Algorithms with Structured and Panmictic Islands" *IPDPS 1999 Workshop Online Proceedings* [cited 2003 Dec 1] Available at HTTP: http://ipdps.eece.unm.edu/1999/biosp3/alba.pdf

[8] T. Myer, IBM, "Grid computing: Conceptual flyover for developers"[Online Document] May 2003 [cited 2003 Dec 1] Available HTTP: http://www-106.ibm.com/developerworks/grid/library/gr-fly.html?ca=dgr-lnxw01GridFlyover

[9] SETI@Home, "SETI@Home" [Online Document] [cited 2003 Dec 1] Available HTTP: http://setiathome.ssl.berkeley.edu/

[10] FightAIDS@Home, "FightAIDS@Home" [Online Document] [cited 2003 Dec 1] Available HTTP: http://fightaidsathome.scripps.edu/

[11] G. S. Amalsi and A. Gottlieb, *Highly Parallel Computing* (2nd edition), Benjamin/Cummings, 1994.

[12] J. Gustafson, "Fixed Time, Tiered Memory, and Superlinear Speedup". Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5), October 1990.

[13] E. Alba, A. Nebro, and J. Troya, "Heterogeneous Computing and Parallel Genetic Algorithms". Journal of Parallel and Distributed Computing 62, 1362–1385 2002

[14] G. Booch, *Object-oriented analysis and design with applications*. Addison-Wesley: Menlo Park, CA, 1994.

[15] (author unknown) "PGAPack: Parallel Genetic Algorithm Library" [Online Document] [cited 2003 Dec 1] Available HTTP: http://www-fp.mcs.anl.gov/CCST/research/reports_pre1998/comp_bio/stalk/pgapack.html

[16] B. Paechter, "The DREAM Project" [Online Document] [cited 2003 Dec 1] Available HTTP: http://www.world-wide-dream.org/

[17] P. Roy, and B. Abdulhai, "GAID: Genetic Adaptive Incident Detection for Freeways", Transportation Research Board 2003.

Appendix A: Using a Galapagos Application

(The following pages form the basic starter documentation for users of Galapagos, and applies to the Galapagos build 10. Technical documentation is provided as JavaDoc files with the Galapagos distribution.)

Introduction

This document provides most of the information a user needs to perform a Galapagos GA run. These instructions are written so as to apply generally to most platforms/setups, but some note on specific network configurations are included as an appendix, and could be useful to users.

Going beyond this introductory document, the Galapagos operator documentation (in doc/index.html) provides explanation of the provided meta-operators and supporting operators and a few examples as to typical configurations. The Galapagos and LightGrid developer documentation provides more technical information as to how to write your own operators or extend the evaluator and controller components.

Requirements

Before trying to follow these instructions, be sure that the latest version of the Sun Java Runtime Environment (JRE) is installed on all the computers you are planning to use and that they can all 'see' each other on the network correctly, you will also need each computer's local hostname on hand. This may seem obvious, but you will also need some sort of access (physical or remote) to the machines, with the correct passwords and permissions to read, write and execute files, use the JRE etc. Please consult with the local system administrator regarding this if you are unsure.

Galapagos takes the form of a '.jar' file which contains all of the LightGrid and Galapagos components. This document presumes that you have on hand this file and the extended components that you intend to use, compiled and properly packaged into a jar-file or class-files in a packaged directory structure, which is rooted (located) in the same directory as the Galapagos jar-file. If your arrangement is different, some of the commands listed in this document will have to be modified accordingly.

The information you need to be able to code such extended components is contained in another document, but the short version is that Galapagos provides some abstract classes (or programming templates), which must be extended by another developer before Galapagos can be used as a GA platform to solve a given problem. Specifically, the Controller and Evaluator components are specific to each problem/application.

Necessary Files

To invoke the executable components of the Galapagos jar-file, one must use the command-line from the same directory as the jar-file. This can be automated or simplified by storing the relevant commands in batch ('.bat') files on Windows machines, or shell scripts ('.sh' or '.csh' files) on Linux or Unix clones These files can then be called from the command line (or by double-clicking on it in a windowing environment). From the command-line, you may use the following commands to launch the various components of Galapagos. NOTE: on a non-Windows machine, the semicolons (;) in the following table must be changed to colons (:).

Controller	`java –cp galapagos.jar;lightgrid.jar;[classpath to your code] [fully qualified name of your controller]`
Dispatcher	`java –cp lightgrid.jar ca.utoronto.civ.its.lightgrid.dispatcher.Dispatcher`
Container	`java –cp galapagos.jar;lightgrid.jar[;classpath to any custom operators] ca.utoronto.civ.its.galapagos.container.Container`
Evaluator	`java –cp galapagos.jar;lightgrid.jar;[classpath to your code] ca.utoronto.civ.its.lightgrid.resource.Resource`

If you wish to use shell scripts or batch files, simply create a text file, i.e. one called 'dispatcher.txt', open it the usual way, paste the right command from the above table, close it then change the filename to 'dispatcher.bat' or 'dispatcher.sh' as appropriate, ignoring any warnings the OS may throw about changing file extensions.

The various Galapagos components also require that certain files be in the same directory as the jar-file and the scripts/batch files/issued command-line commands. They are listed in the table below, and in the back of this document is a table summarizing this page.

Controller	*config.xml* – the file which contains all of the configurations for the GA run, and is explained below *logs* – a directory to store logs in
Dispatcher	*resources.txt* – an optional file which contains the hostnames of any evaluators available to the controller prior to its launch
Container	*dispatcher.txt* – a file which contains only the hostname of the computer hosting the dispatcher *controller.txt* – file which contains only the hostname of the computer hosting the controller
Evaluator	*dispatcher.txt* – an optional file which contains only the hostname of the computer hosting the dispatcher

XML files and config.xml

This portion of this document details how to write a config.xml file for your Galapagos run. This file is an eXtensible Markup Language (XML) document (with certain caveats) and it is very easy to pick up XML syntax. XML essentially allows people to store and pass around data in text files in a structured manner. XML files are meant to be both machine-readable and human-readable. The data is 'marked-up' or 'tagged' by the usage of special text strings: tags. An XML fragment looks like this:

```
<tag> data data data </tag>
```

Any text enclosed in angle brackets is a tag. Any text not enclosed in angle-brackets is data. A tag is 'opened' when it is placed. It is 'closed' when it next appears with a slash (/) before its name, as above. Any data between the two is 'contained' by the tag. Tags may be nested (they may contain other tags), but may not overlap, as follows: the first example is ok, the second is not:

```
<tag1> data <tag2> data </tag2> data </tag1>                    CORRECT
```

```
<tag1> data <tag2> data </tag1> data </tag2>                    INCORRECT
```

A technique used below is to represent an XML file as a tree, or nested list, as follows:

```
<tag1>
  <tag2> data1 </tag2>
  <tag3> data2 </tag3>
  <tag4>
    <tag5> data3 </tag5>
  </tag4>
</tag1>
```

- tag1
    - tag2

        - data1
    - tag3
        - data2
    - tag4
        - tag5
            - data3

XML files are very simple to create: they are text files with the extension '.xml'. You may create/edit such files in the same manner as the batch files or shell scripts outlined previously.

XML files can be viewed on Windows machines with Internet Explorer, which provides some handy formatting functions.

The easiest way to both explain and understand the config.xml file (and how Galapagos basically functions) is to dissect a sample config.xml, which is on the next page. Note that the formatting of this file (the placement of new lines and indentation) is simply for clarity, with the exception that, unlike regular XML, there *must* be some sort of whitespace (newline, tab, or space) around every tag.

A Sample config.xml

```
<?xml version="1.0"?>

<parameters>
  <notes> This is a note </notes>
  <fitness>
    <name> ca.utoronto.civ.its.gaid.GAIDFitness </name>
    . . .
  </fitness>
  <convergence>
    <name>
      ca.utoronto.civ.its.galapagos.controller.convergences.StdConvergence
    </name>
    <minstd> 0.001 </minstd>
  </convergence>
  <chromosome>
    <name>
      ca.utoronto.civ.its.galapagos.chromosomes.RealChromosome
    </name>
    <numvars> 16 </numvars>
    <geneminvalues>
      <gene1> 0.001 </gene1>
      .
      .
      .
      <gene16> 0.001 </gene16>
    </geneminvalues >
    <genemaxvalues>
      <gene1> 1000.0 </gene1>
      .
      .
      .
      <gene16> 1000.0 </gene16>
    </genemaxvalues >

  </chromosome>
  <container>
    <containerid> theContainer </containerid>
```

```
<containerhostname> p1.ecf.utoronto.ca </containerhostname>
<dispatcherhostname> p2.ecf.utoronto.ca </dispatcherhostname>
<population>
  <populationid> thePopulation </populationid>
  <populationsize> 50 </populationsize>
  <generationsize> 50 </generationsize>
  <sendatonce> 50 </sendatonce>
  <initializer>
    <name>
    ca.utoronto.civ.its.galapagos.operators.initializers.RandomInitializer
    </name>
  </initializer>
  <generator>
    <name>
    ca.utoronto.civ.its.galapagos.operators.generators.StandardGenerator
    </name>
    <recombiner>
      <name>
      ca.utoronto.civ.its.galapagos.operators.recombiners.RandomCrossover
      </name>
      <selector>
        <name>
        ca.utoronto.civ.its.galapagos.operators.selectors.RandomSelector
        </name>
      </selector>
    </recombiner>
    <mutator>
      <name>
  ca.utoronto.civ.its.galapagos.operators.mutators.SingleGeneUniformMutator
      </name>
      <genemutator>
        <name>
      ca.utoronto.civ.its.galapagos.operators.genemutators.CreepGeneMutator
        </name>
        <mutationrate> 400 </mutationrate>
      </genemutator>
    </mutator>
  </generator>
  <assembler>
    <name>
```

```
        ca.utoronto.civ.its.galapagos.operators.assemblers.SimpleAssembler
        </name>
      </assembler>
      <migrator>
        <name>
        ca.utoronto.civ.its.galapagos.operators.migrators.NullMigrator
        </name>
      </migrator>
    </population>
  </container>
</parameters>
```

Sample config.xml Dissected

The first line of the sample config.xml is required at the top of all XML documents. It is not of much interest, other than that it must be there, verbatim, on all config.xml files given to Galapagos.

What follows is a tree-view of the rest of config.xml, with explanations (optional parts are in italics).

- parameters – this is the container tag for the whole file
    - *... notes – whatever data is here will simply be copied into the log*
    - fitness
        - name – this is the fully qualified class name for the Fitness
        - *.. other tags*
    - convergence – this contains data for the when the run will stop
        - name – fully qualified class name
        - *... other tags*
    - chromosome – this contains data relevant to the problem representation
        - name – this is the fully qualified class name for the Chromosome class
        - numvars – the number of genes
        - minvalues – the minimum gene values
            - *... gene by gene values*
        - genemaxvalues – the maximum gene values
            - *... gene by gene values*
    - container – this contains data relevant to a single container component
        - containerid – this is a unique id
        - containerhostname – this is the hostname of the computer hosting this container
        - population – this contains data relevant to a single population or deme
            - populationid – this is a unique id
            - populationsize – the number of chromosomes in this deme
            - generationsize – the generation size for this deme
            - sendatonce – the number of chromosomes to be sent for evaluation at once (experimental parameter, usually equals generationsize)
            - initializer – this contains data for the initializer operator
                - name – fully qualified class name
                - *... other tags*
            - generator – this contains data for the generator operator
                - name – fully qualified class name
                - *... other tags*
            - assembler – this contains data for the assembler operator
                - name – fully qualified class name
                - *... other tags*

- migrator – this contains data for the migrator operator
  - name – fully qualified class name
  - *... other tags*

Note that the optional parts listed above are the 'notes' subtag of 'parameters' and all of the subtags of 'initializer', 'generator', 'assembler' and 'migrator' except for 'name'. The latter will be explained presently.

As mentioned earlier, the basic Galapagos controller logs a variety of information in its log files. A component of this information is the complete contents of config.xml, which makes it handy to re-run the same configuration, and makes for easier record-keeping. The 'notes' field above is not recognized by the controller as meaningful, so it is merely copied verbatim (minus indentation/newlines) into the logfile. This means that you could use any tag within the 'parameters' container tag to store whatever meta-data you wish. You may not place such tags anywhere else, though.

The 'chromosome' tags are straightforward GA parameters, as are 'populationsize' and 'generationsize'. 'sendatonce' is an experimental parameters in this release and should be set to equal 'generationsize' for most applications. 'fitness' refers to the Java class that is to be used in this GA. 'convergence' defines when the GA run should stop.

In the sample above, there is one 'container' tag and one 'population' tag, but the 'parameter' tag can contain as many 'container' subtags as required and the 'container' tag can contain as many 'population' tags as necessary. This is how Galapagos allows for any mapping of deme-to-computer in a PGA. Note that the order in which you place the 'container' tags governs the order in which you launch the containers, as the parameters are assigned in order, so if this matters to your application (i.e. you have structured the deme or container settings to be computer-specific) be careful in the order in which you launch the containers.

The 'containerid' and 'populationid tags must be unique (i.e. you may not have the same containerid twice or the same populationid twice) and are there simply for reference, both internally to Galapagos, and in logs and command-line prompts.

The 'containerhostname' tag tells the system where each container runs, and these processes must be launched before the file is loaded (see later in this document). The 'dispatcherhostname' tag tells the system where each container will find its dispatcher. If left unspecified, the container will use whatever it found in dispatcher.txt.

The body of the 'population' element configures Galapagos' operators. In designing Galapagos, various GA types were analyzed, abstracted and decomposed into independent modular units: operators. Galapagos provides some common operators as a library and further provides the templates needed to write new operators as needed. Each population in Galapagos relies on four meta-operators:

- initializer – this operator governs which chromosomes will be in the initial population, an example would be one that seeds the population with random chromosomes

- generator – this operator governs which chromosomes will be in each generation, and includes such operations as selection, crossover and mutation, but can also contain ES recombinations etc
- assembler – this operator governs how each generation interacts with the population at each iteration to form the new population
- migrator – this operator governs the migration epoch, topography, selection and replacement strategies in a PGA

Beyond the four meta-operators, Galapagos defines other operators to be used designing GAs. For example, while there are various ways to generate a generation, most will include selection, mutation or crossover or recombination. Migration also requires selection rules to determine which chromosomes are sent and how they integrate into the receiving population. The other, supporting, operators defined by Galapagos are as follows:

- selector – given a set of chromosomes, a selector will select a given number according to some rule
- recombiner – takes some chromosomes (usually defined by a selector operator), returns a child, according to some rule
- mutator – takes a chromosome, returns it mutated according to some rule (usually mutated by a genemutator) mutators are intended to govern *which* genes are mutated
- genemutator – takes a gene, mutates it according to some rule, genemutators are intended to govern *how* a given gene is mutated
- epoch – defines a time period
- topography – governs which populations migrants will be sent to and how many will be sent

The way in which the meta-operators interact with the supporting operators largely depends on which meta-operator is in question, but by and large, the follow statements apply:

- generator – uses a mutator and a recombiner
- assembler – uses a selector
- migrator – uses a selector, an assembler, an epoch and a topography

Some of the supporting operators can also rely on each other, usually as follows:

- recombiner – uses a selector
- mutator – uses a genemutator

When an assembler ‘uses’ a selector, any selector may be used, from the same pool of available selectors as the ones from which a recombiner’s selector or a migrator’s selector may be chosen. The way in which these various settings are assigned is via the optional parameters in config.xml.

The only required subtag of the meta-operator tags is ‘name’ which must contain a valid fully qualified class name to an operator of the correct type (as outlined in the Galapagos operator documentation) for example:

```
ca.utoronto.civ.its.galapagos.operators.selectors.RouletteWheelSelector
```

The optional subtags depend on the parameters needed by the operator at hand, and are all detailed in the Galapagos operator documentation, or by whoever wrote the operator you are trying to use. The following paragraphs explain how to specify these extra parameters by looking at the sample config.xml.

The initializer specified is the Galapagos-provided RandomInitializer, which takes no parameters (as outlined in the operator documentation) so there are no subtags other than name, which specifies this operator with a fully qualified class name.

The generator specified is the Galapagos-provided StandardGenerator, which, according to the documentation, requires a mutator and a recombiner. These operator tags are simply placed inside the ‘generator’ tag. This is how config.xml specifies which supporting operators to use. The mutator specified requires a genemutator, which is also specified and itself requires a numeric parameter, which is again specified using an XML tag containing some data. The recombiner requires a selector, which is specified, and the one chosen does not require any further parameters, so it does not have any subtags other than ‘name’.

The assembler specified is the Galapagos-provided SimpleAssembler, which takes no parameters.

The migrator specified is the Galapagos-provided NullMigrator, which is simply a placeholder that says ‘no migration’ as there is only a single population (the way it does this is by using an internal epoch which never expires, so migration never occurs).

By now, it should be clear how to specify various operators, supporting operators and parameters. Hopefully, the flexibility of this approach is also clear, in that the GA structure can be specified to a high granularity and that operators can be used, reused, swapped etc. The fact that anyone can write and immediately use new operators is also significant. The way in which this would be done would simply be to specify an operator not in the Galapagos package but somewhere else in the classpath. It is likely that most Galapagos applications will require their own initializer operator to correctly seed the population.

Beyond flexibility in operators, it should also be clear now that Galapagos provides flexibility in terms of populations (or demes) in that each deme can be the same or radically different and that the user has total control over which computer hosts which or how many demes etc.

More information about specific operators can be found in the operator Javadoc documentation which should be available with this document.

Steps to Launch a Galapagos Run

This portion of the documentation assumes you have correctly set up all of the files listed above, and have a correctly formatted config.xml file. Please read the rest of this document if you need to learn how to do the latter. Note that the steps listed below are order-sensitive, it is indicated where order is optional. Further note that the word 'launch' is intended to mean 'invoke command on command-line' or 'call script' or 'double-click on script' depending on what setup you have opted to use.

1. ensure that you have a complete plan of what you wish to do and that config.xml matches this, including what machines are available to you, and on which machine each component is intended to run etc, also ensure that there is no scheduled downtime on each of these machines
2. launch evaluators (note: this can happen again at any time during the run, but any that are launched before the dispatcher is launched must be listed in resources.txt)
3. launch dispatcher
4. launch the containers listed in config.xml (these could simply be on all the time)
5. [on the dispatcher machine] ensure that all evaluators (resources) have been registered
6. launch controller

7. [on the controller machine] ensure that config.xml is correct, hit 'load'
8. [on the controller machine] ensure that all parameters have been sent
9. [on the container machine(s)] ensure that Galapagos parameters have been loaded
10. [on the dispatcher machine] ensure that all populations (clients) have been registered
11. [on the controller machine] at your leisure, you may now hit 'start'

Galapagos will now run your GA.

12. [on the controller machine] monitor the state of your GA run as appropriate, you may hit 'reset' as needed.
13. [on the controller machine] if/when the GA terminates, either by converging or by your reset, you are brought back to the pre-load step, where you may modify config.xml, then hit load. Assuming that the evaluators and containers are still active, this basically starts a new run right away.

14. [on the controller machine] to shut down the containers, hit 'killcontainers', then hit 'quit' to shut down the dispatcher (this can only happen when it is prompting you for a 'load' command, i.e. after a 'reset')
15. [on the dispatcher machine] to shut down the evaluators, hit 'killresources', then hit 'quit' to shut down the dispatcher

Troubleshooting

If Galapagos behaves strangely or crashes, here are the steps you should try:

- check to make sure all launch commands are correct and the classpaths are reasonable
- check to make sure that all Galapagos and LightGrid components have been launched in the right sequence and that they have all ‘found’ each other
- check to make sure the config.xml file is correctly formatted (spaces around each tag)
- check to make sure the config.xml file has all the required elements
- check to make sure the config.xml file has elements in the correct order
- check to make sure your application code is not responsible
- check to make sure all network connections and permissions are set correctly

Typical Directory File Structure

All of the following files should be in the same directory. The name of this directory does not matter, but is ‘galapagos’ in this example.

- galapagos/
    - galapagos.jar
    - lightgrid.jar
    - config.xml (on controller machine)
    - dispatcher.txt (on container and evaluator machines)
    - resources.txt (on dispatcher machine)
    - controller.txt (on container machines)
    - [batch or script files]
    - [application-specific jar or class files]
    - [any supporting files needed by the application]
    - logs/ (on controller machines)
        - [Galapagos will store logs here by default]

Notes on Specific Network Types

NFS (network file system) setups:

- Galapagos was developed on an NFS system
- dispatcher and controller automatically write their hostname to the txt files, and on an NFS system, these are visible to the other computers, making things quicker to set up

Systems where remote login is available:

- if SSH or telnet is available, scripts can be written to automate launching the various containers or evaluators from one machine

Systems where the administrator is friendly:

- you can install the containers and evaluators as ‘services’ on Windows or ‘daemons’ on Unix/Linux and they will always be available. As long as application-specific code is in the classpath, you should be able to tap these processes at will.