A DISTRIBUTED POOL ARCHITECTURE FOR GENETIC ALGORITHMS

A Thesis

by

GAUTAM SAMARENDRA N ROY

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2009

Major Subject: Computer Engineering

A DISTRIBUTED POOL ARCHITECTURE FOR GENETIC ALGORITHMS

A Thesis

by

GAUTAM SAMARENDRA N ROY

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Co-Chairs of Committee,	Jennifer Welch
	Nancy Amato
Committee Members,	Takis Zourntos
Head of Department,	Valerie Taylor

December 2009

Major Subject: Computer Engineering

ABSTRACT

A Distributed Pool Architecture for Genetic Algorithms. (December 2009)

Gautam Samarendra N Roy, B. Tech., Indian Institute of Technology Guwahati

Co–Chairs of Advisory Committee: Dr. Jennifer Welch
Dr. Nancy Amato

The genetic algorithm paradigm is a well-known heuristic for solving many problems in science and engineering in which candidate solutions, or “individuals”, are manipulated in ways analogous to biological evolution, to produce new solutions until one with the desired quality is found. As problem sizes increase, a natural question is how to exploit advances in distributed and parallel computing to speed up the execution of genetic algorithms. This thesis proposes a new distributed architecture for genetic algorithms, based on distributed storage of the individuals in a persistent pool. Processors extract individuals from the pool in order to perform the computations and then insert the resulting individuals back into the pool. Unlike previously proposed approaches, the new approach is tailored for distributed systems in which processors are loosely coupled, failure-prone and can run at different speeds. Proof-of-concept simulation results are presented for four benchmark functions and for a real-world Product Lifecycle Design problem. We have experimented with both the crash failure model and the Byzantine failure model. The results indicate that the approach can deliver improved performance due to the distribution and tolerates a large fraction of processor failures subject to both models.

To my parents

TABLE OF CONTENTS

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION*

Genetic algorithms (GAs) are powerful search techniques for solving optimization problems [1, 2]. They are inspired by the theory of biological evolution and belong to the class of algorithms known as evolutionary algorithms. These algorithms provide approximate solutions, and are typically applied when classical optimization methods cannot be used or are too computationally expensive.

In genetic algorithms a population of abstract representations of candidate solutions ("individuals" or "chromosomes") evolves towards better solutions over multiple "generations". The algorithm begins with a population of (typically random) individuals. At each iteration, the individuals are evaluated using a fitness function to select a subset. The chosen individuals are given the opportunity to "reproduce" (create new individuals) through two stochastic operators, mutation and crossover, in such a way that the better solutions have greater chance to reproduce than the inferior solutions. Crossover cuts individuals into pieces and reassembles them, while mutation makes random changes to an individual. A genetic algorithm normally terminates when a certain number of iterations has been performed, or a target level of the fitness function is reached by at least one individual. The candidate solution encoding and fitness function are dependent on the specific problem to be solved.

This thesis follows the style of *IEEE Transactions on Evolutionary Computation.*

As problem sizes increase, a natural question is how to exploit advances in distributed and parallel computing to speed up the execution of genetic algorithms. This thesis proposes a new distributed architecture for genetic algorithms, based on distributed storage of candidate solutions ("individuals") in a persistent pool, called Pool GA. After initializing the pool with randomly generated individuals, processors extract individuals from the pool in order to perform the genetic algorithm computations and then insert the resulting individuals into the pool.

Unlike previously proposed approaches, the new approach is tailored for loosely coupled, heterogeneous, distributed systems and works well even in the presence of failures of components. Since individuals can be stored separately from GA processors, the failure of a processor does not cause good individuals to be lost. Also, the individuals can be replicated for additional fault tolerance.

We have simulated the Pool GA approach on a variety of applications using simple selection, crossover and mutation operators, in order to obtain some proof-of-concept results. Four of the application problems are continuous functions drawn from the literature [3] and are considered good benchmark problems for testing GAs. The results show that there is a clear advantage using concurrent processing in that the same level of fitness is achieved faster with more processors.

We also apply our approach to a real-world Product Lifecycle Design problem. Product Lifecycle Design involves planning ahead to reuse or remanufacture certain components to recover some of their economic value. A recently developed decision model [4] indicates that component reuse and remanufacture can simultaneously decrease cost and increase customer satisfaction; however, computational issues have prevented the scaling of the analysis to larger, more realistically sized problems. New computational methods, such as distributed approaches, therefore need to be considered that can quickly and reliably determine the optimal solution, thus allowing exploration of more of the design space.

Having the capability to quickly and efficiently solve the optimization problems allows re-running the code under varying input conditions. It allows for evaluating scenarios before they occur and formulating strategies for different design conditions. As new insights are gained, products can be redesigned and enhanced quickly with minimal deviations from optimality under changing conditions. We have applied our Pool GA to a simple version of this problem. The results look promising and we expect that more realistic versions of the problem will benefit even more from our distributed approach.

We have simulated two types of processor failures in testing our Pool GA. In the crash failure model, the failing processors simply stop at an arbitrary instant. In the Byzantine failure model, introduced by Lamport et al. [5], the faulty processors can exhibit arbitrary deviation from their expected behavior. This failure model is thus more malignant than the crash failure model. The Byzantine processors can, for instance, independently write back poor fitness individuals into the pool, or several Byzantine processors could try to cooperate and try to delay the progress of the GA. In general the Byzantine failure model captures the faulty behavior that is the worst for the algorithm.

There are thus many ways in which Byzantine processors may be simulated. We simulate Byzantine behavior by what we call Anti-Elitism in which the Byzantine processors continue to run the GA algorithm as before; however, they write back a new individual to the pool only if it is worse than the existing individual in the pool. We call it Anti-Elitism, because this behavior is the exact opposite of the GA concept of elitism, wherein new individuals are considered for further reproduction only if they are better than the individual from the previous generation. The simulation results indicate that the algorithm is tolerant to a high percentage of processor failures of both crash and Byzantine type.

A preliminary version of the results in this thesis appeared in [6].

CHAPTER II

RELATED WORK*

Whitley [2] provides a good starting resource for the study of genetic algorithms. He also summarizes some theoretical foundations for genetic algorithms based on the arguments of Hyperplane Sampling and the Schema Theorem and gives some insight as to why genetic algorithms work. Many theoretical advances have also been made in recent times to further the understanding of genetic algorithms as enumerated by Rowe in [7].

Advances in computing technology have increased interest in exploring the possibility of parallelizing genetic algorithms. Prior proposals for distributed or parallel genetic algorithms can be classified into three broad models, the Master-Slave model, the (coarse grained) Island model, and the (fine grained) Cellular model [2].

In the Master-Slave model, a master processor stores the population and the slave processors evaluate the fitness. The evaluation of fitness is parallelized by assigning a fraction of the individuals to each of the processors available. The algorithm runs synchronously in that the master process waits to receive the fitness values of all individuals before proceeding to the next generation. Communication costs are incurred whenever the slaves receive individuals to evaluate and when they return back the fitness values. Apart from evaluating the fitness, another part of the GA that can be parallelized is the application of mutation and crossover operators; however these operators are usually very simple and the communication cost of sending and receiving individuals will normally offset the performance gain by

parallelization. In summary, the Master-Slave model has advantages when evaluating the fitness of the individuals is time-consuming. If a slave fails in the Master-Slave model, then the master may become blocked. In our Pool GA approach, the algorithm is not stalled due to the failure of a participating processor.

In the Island model, the overall population is divided into subpopulations of equal size, the subpopulations are distributed to different processors, and separate copies of a sequential genetic algorithm are run on each processor using its own subpopulation. Every few generations the best individuals from each processor "migrate" to some other processors [8]. The migration process is critical to the performance of the Island model. Of great interest is to understand the role of migration on the performance of this parallel GA, such as the effect of frequency of migration, the number of individuals exchanged each time, the effect of communication topology, etc. Cantú-Paz [8] discusses some of the past work on this subject and also states that most of these problems are still under investigation. Another open question is to find the optimal number of subpopulations to get the best performance in terms of quality of solutions and speed of convergence. The interaction between processors is mostly asynchronous; the processors do not wait for other processors to take any steps. The failure of a processor in the Island model can cause the loss of good individuals. In our Pool GA approach, all individuals computed are available to the other processors even after the generating processor fails.

In the Cellular GA model, also known as fine-grained GA or massively parallel GA, there is one overall population, and the individuals are arranged in a grid, ideally one per processor. Communication is restricted to adjacent individuals and takes place synchronously.

Recently, there has been interest in developing parallel GAs for multi-objective optimization problems. Deb et al. [9] provide a parallel GA algorithm designed to find the Pareto-Optimal solution set in multi-objective problems. Their algorithm is based on the

Island model.

The idea of keeping the candidate solutions for the genetic algorithm in a “pool” was inspired by the Linda programming model [10, 11], and has also been used by others (e.g., [12, 13]). Sutcliffe and Pinakis [12] embedded the Linda programming paradigm into the programming language Prolog and mentioned, as one application of the resulting system, a genetic algorithm in which candidate solutions are stored as tuples in the Linda pool and multiple clients access the candidate solutions in parallel. In contrast to our thesis, no results are given in [12] regarding the behavior of the parallel GA. Davis et al. [13] describe a parallel implementation of a genetic algorithm for finding analog VLSI circuits. The algorithm was implemented on 20 SPARC workstations running a commercial Linda package. Two versions of the algorithm are presented: the first one follows the Master-Slave model and the second one is a coarse-grained Island model in which each of the four islands runs the Master-Slave algorithm. In contrast, our algorithm is fine grained, and we evaluate the behavior of the algorithm through simulation with varying numbers of processors.

In [14], a distributed GA is proposed that uses the Island model and a peer-to-peer service to exchange individuals in a message-passing paradigm. In contrast we use a more fine-grained approach than the Island model and use a shared object paradigm for exchanging individuals between processors, and we provide more extensive simulation results.

The candidate solutions in our approach are examples of distributed shared objects (e.g., [15]). They can be implemented using replication (e.g., [16]). Previous work has suggested such approaches for other aspects of the Product Lifecycle Design problem [17].

Hidalgo et al. [18] studied the fault tolerance of the Island model in a specific implementation with 8 processors subject to crash failures. Their results suggest that, at least for multi-modal functions, there is enough redundancy among the various processors for there to be implicit fault tolerance in the Island model. One of their conclusions is that it is better

to exchange individuals more frequently than to have a large number of islands. Lombrana et al. [19] came to similar conclusions about the inherent fault-tolerance of parallel GAs based on simulations of a Master-Slave method. Our results can be considered an extension to the case of fine-grained parallelism, in which individuals are exchanged all the time and each processor is an island. Furthermore, in our approach, since individuals are stored separately from GA processing elements, they can be replicated for additional fault tolerance so that the failure of a processing element does not cause good individuals to be lost.

Merelo et al. [20] proposed a framework using Ruby on Rails to exploit spare CPU cycles in an application-level network (e.g., SETI@Home) using a web browser interface. Experiments were done with a genetic algorithm application in which the server was the master and volunteer "slave" nodes could request individuals to evaluate.

The work reported in this thesis was originally motivated by attempts to find computationally efficient solutions to large instances of the Product Lifecycle Design problem. Modeling of the entire lifecycle of a product is widely advocated for environmentally benign design and manufacturing. Product Lifecycle Design aims to reduce the environmental impact over the entire lifecycle. For example, Kimura [21] proposed a framework for computer support of total lifecycle design to help designers performing rational and effective engineering design. Pandey and Thurston [22] applied the Non-dominated Sorting Genetic Algorithm (NSGA-II) to identify non-dominated solutions for component reuse in one lifecycle. A service selling (leasing) approach can also be envisioned where the manufacturer retains the ownership of the product and upgrades the product when considered necessary or if desired by the customer. Mangun and Thurston [4] developed such a decision model indicating that a leasing program allows manufacturers to control the take-back time, so components can be used for multiple lifecycles more cost-effectively. Sakai et al. [23] proposed a method and a simulation system for Product Lifecycle Design based on product life control.

CHAPTER III

THE POOL GA ARCHITECTURE*

In the proposed Pool GA Architecture, there are multiple processors, each running a copy of the GA. Unlike the Island model, each processor is not confined to a set of individuals: there is a common pool of individuals from which each processor picks individuals for computing the next generation. The pool size is larger than the population of the individual GA working on each processor. Thus, our Pool GA model can be viewed as an Island model with migration frequency of one per generation and the number of individuals allowed to migrate is equal to the population size of the GA.

We now describe the working of the Pool GA Architecture in detail.

There are $p \geq 1$ participating processors. Each participating processor runs a sequential GA with a population of size u. There is a common pool $\mathcal{P}$ of individuals of size $n > u$. Each individual in the pool is stored in a shared data structure, which can be accessed concurrently by multiple processors. There is a rich literature on specifying and implementing shared data structures (e.g., [24]). For the current study, we have chosen to store each individual as a multi-reader single-writer register. In more detail, $\mathcal{P}$ is partitioned into $\mathcal{P}_1, \ldots, \mathcal{P}_p$. Each partition $\mathcal{P}_k (1 \leq k \leq p)$ is a collection of single-writer (written by processor k), multi-reader (read by any of the p processors) shared variables where each shared variable holds an individual of the GA. Initially the individuals in $\mathcal{P}$ are randomly generated.

There are two basic operations performed on $\mathcal{P}$ by any participating processor: ReadIn and WriteOut. The ReadIn operation performed on $\mathcal{P}$ by processor k picks u individuals uniformly at random from $\mathcal{P}$ and copies them into k's local data structure P_k. The WriteOut operation performed on $\mathcal{P}$ by processor k writes back the individuals in P_k to the portion of $\mathcal{P}$ that is allotted to k. Here, in order to ensure convergence of the GA, an element of elitism is applied, i.e. the individual i in P_k replaces an individual j in $\mathcal{P}_k$ only if i is fitter than j. (Other schemes are possible; this one was chosen for concreteness.)

Between the ReadIn and WriteOut operations, each processor k performs a local procedure Generate to generate a new generation of individuals from the individuals in P_k. The Generate procedure consists of Selection, Crossover and Mutation operations. The choice of these operators is up to the implementer and based on the problem. The operators in our simulation are described in the next chapter.

One of the design goals of the Pool GA Architecture was to enable processors with different speeds to participate together in the GA and improve tolerance to failures of some of the participating processors. The Pool GA achieves both these goals by decoupling the operation of processors from each other: i.e., the processors interact with only the pool and are unaware of each other's existence. Processors do not explicitly synchronize with each other and can be working on different generations at the same time.

An important part of any GA is the method of termination. There are various termination criteria that may be used in conjunction with our Pool GA. For the scenario where the desired fitness level is known, once any processor discovers an individual with that fitness it can terminate. It can also inform the other processors before terminating, so that they can also terminate. The above method takes advantage of differences in processor speeds. In the case where the desired fitness level is unknown a couple of strategies can be used. One is to let the GA run for a sufficient predecided number of generations and then terminate. Another is to let a processor terminate once it sees very small change in the best fitness

value generated for few continuous generations.

The Pool GA Architecture could support a dynamically changing set of participating processors, as it provides persistent storage for individuals independent of the processors that created them. A possible advantage of such a loosely coupled asynchronous model is that large problems can be solved in a distributed fashion: users worldwide can volunteer the free time on their computers for processing the problem. The Berkeley Open Infrastructure for Network Computing [25] gives a list of many such projects using distributed computing over the Internet.

It is important to note that the Pool GA Architecture is termed as an "architecture" and not an algorithm because it is not tied to specific selection, crossover or mutation operators. It gives a paradigm for maintaining a large set of potential solutions and defines a procedure by which multiple processors can cooperatively solve the GA problem by accessing a pool of individuals.

We believe the Pool GA Architecture can provide more fault tolerance than the existing models. In the Island model if a processor fails, the individuals it holds are lost with it. In the unfortunate case where the fittest individual was located at that failed processor, that individual could be lost and convergence would be delayed. If a slave fails in the Master-Slave model, then the master may become blocked; moreover, the master is a single point of failure for the entire algorithm. In the Pool Architecture, failures of the processors cannot lead to loss of individuals, since individuals are stored separately from processors, and they do not cause the algorithm to block since the correct processors continue to operate. In contrast in our case as the pool is decoupled, even if a processor which found a good individual fails, other processors will have access to that individual. The pool is not a single point of failure (like the master is) because fault-tolerance for the individuals can be achieved using standard distributed computing techniques with replication and quorum systems (e.g., [16]).

CHAPTER IV

IMPLEMENTATION*

We simulated our Pool GA with a C++ program written in the POSIX multi-threaded environment. In the simulation each POSIX thread represents a processor participating in the Pool GA. The simulation can be easily modified to use OpenMP or other parallel programming paradigms for multiprocessors when the hardware is available. The simple GA code in C provided at the KANGAL website [26] was adapted to a multi-threaded version. We used the operators available in the KANGAL code. A tournament-based selection operator is used for selection. For discrete-valued problems ("binary GAs"), a single point crossover operator was used, and the mutation operator flipped each bit of the individual with the probability of mutation. For real-valued problems ("real GAs"), the Simulated Binary Crossover (SBX) operator and the polynomial mutation operator were used. These operators are not tied in any way to the Pool Architecture and can easily be changed according to the problem.

The common pool of n individuals which are possible solutions to our distributed GA is represented in the code by a shared global array of length n. Let u be the per-thread population size. The threads (each representing one processor in the real scenario) run their own GA algorithm on a subset of the pool. In each generation, a thread uses ReadIn to pick u random indices from the array, which act as its current population. The thread performs Selection, Crossover and Mutation on these individuals and generates the next

generation. This new generation is written back to the pool at specific indices based on the thread id using the WriteOut operator. For WriteOut, the array representing the pool is considered to be partitioned into p segments, where p is the number of threads, each of size u. Each thread can read from any element of the array, but can only write to its own partition. More specifically, after computing u new individuals, $c_1, c_2, \ldots, c_u$, the WriteOut operator on the pool is implemented by having the thread write back each new individual c_i into the i-th entry of the thread's partition if the fitness of c_i is better than that of the current i-th entry. (Alternative ways of implementing ReadIn and WriteOut are of course possible but we did not yet experiment with them.)

Each thread terminates after a certain number of generations. Each thread maintains the best solution it has generated thus far. The overall best solution is picked from among the best solutions of all the threads.

The threads used in the simulation in general behave asynchronously i.e. each progresses independently of others based on the scheduling by the operating system. However in section B of chapter V we present results for synchronous operation of threads, in which each participating thread finishes generation N before any thread begins generation $N + 1$. This lock step behavior is achieved using barrier synchronization in pthreads.

The Pool GA was tested on the following real-valued benchmark minimization functions [3] whose optimal values are given in Table I:

$$
\begin{aligned}
f_1(\vec{x}) &= \sum_{i=1}^{7} 10^{i-1} x_i^2, \\
&\quad -10.0 \leq x_i \leq 10.0 \\
f_2(x_1, x_2) &= 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \\
&\quad -15 \leq x_i \leq 15 \\
f_3(\vec{x}) &= 20 + \sum_{i=1}^{20} (x_i^2 - \cos(2\pi x_i)), \\
&\quad -5.12 \leq x_i \leq 5.12 \\
f_4(\vec{x}) &= \sum_{i=1}^{10} -x_i \sin(\sqrt{|x_i|}), \\
&\quad -500 \leq x_i \leq 500
\end{aligned}
$$

Table I. Benchmark functions and optimal values

Function	Optimum Value
f_1	0
f_2	0
f_3	0
f_4	-4189

We also tested our Pool GA on a Product Lifecycle Design problem, which is a combination of a binary-valued and real-valued problem. This problem is a maximization problem. Background information on the problem and the general mathematical expression of the problem are given in the Appendix. Roughly speaking, the goal is to determine the optimal number of lifecycles for the product (up to a maximum of 8), and within each lifecycle to decide on the optimal choices (of which there are 4) regarding manufacturing

each of the 12 components of the product. Each candidate solution is represented by a $(3 + 8 \cdot 2 \cdot 12) = 195$ bit string.

We have studied the performance of the Pool GA under two fault models: crash and Byzantine. We simulate crash failure of a processor by the exiting of the thread at an arbitrary instant during the execution of the Pool GA. A failure probability is given as a parameter to the simulation. At the start of each generation, a thread tosses a coin with the given probability to decide whether to exit. In case it exits, the thread no longer participates in the GA in any manner.

We simulate Byzantine failures using the Anti-Elitism characteristic. A failure fraction is provided as a parameter to the simulation. For failure fraction f in a simulation with n threads, $\lfloor 100f/n \rfloor$ threads are Byzantine from the outset. Note the difference from our simulation of the crash failures, where the processors crash at varied points during the simulation, while for the Byzantine failure simulations we consider the faulty processors to be Byzantine from the outset. We believe this is more in keeping with the "worst case" notion of the Byzantine failure model.

CHAPTER V

RESULTS*

In this chapter we presents results studying various aspects of the Pool GA using the benchmark problems as well as the Product Lifecycle Design problem. The results relate to

1. The effect of pool size on performance.
2. Speed of convergence as a function of number of threads used.
3. Fault-tolerance to crash and Byzantine failures.
4. Distribution of the fitness values of individuals in the pool at the beginning and end of the Pool GA.

All plots are the average of 10 runs.

A. Effect of Constant Pool Size

Our first simulation experiment compares the performance of a single threaded GA to the performance of our Pool GA with multiple threads while keeping the pool size (i.e., the number of candidate solutions being manipulated) constant. The purpose is to check that the overhead of the parallelism does not cause behavior that is worse than the single-threaded case. Using the lifecycle design problem with the technophile customer group, we

---

compared the performance of the Pool GA for different numbers of threads with a single threaded GA (SGA). In all cases, we used the same algorithm parameters and a fixed pool size of 640. The per-thread population size with t threads was $640/t$.

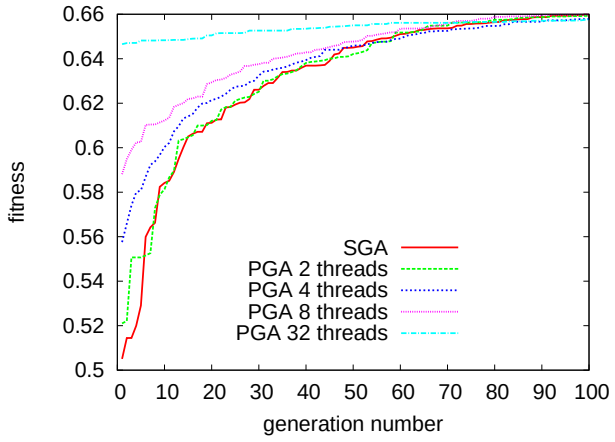

Fig. 1. Lifecycle Design problem for technophile customer group: Speed of convergence over 100 generations with constant pool size of 640

The results are in Fig. 1. All versions of the GA converge to a similar fitness value, indicating that the distribution has not introduced any severe overhead. We also observe that the GA converges faster as the number of threads increases.

However, keeping the pool size constant does not exploit the increased available processing power provided by a distributed GA. Thus in the rest of our simulations, for each problem we keep the population size *per thread* constant, resulting in an overall pool size that increases linearly with the number of threads.

B. Synchronous Operation

We have stated throughout the thesis that the Pool GA architecture is better suited for asynchronous, loosely coupled distributed systems. Before presenting the results corresponding

to asynchronous executions we take a detour and first present results when the processors participating in the Pool GA behave synchronously or in lock step. By synchronous operation we mean that all the processors participating in the GA finish generation N before any processor starts generation $N + 1$. The purposes for showing these results are manifold. Firstly it shows that the Pool GA can work very well even if used in a synchronous manner. Secondly these results clearly show the advantage gained by distributed processing. With more processors the algorithm converges faster and the final fitness values obtained are better. Third, as many existing parallel genetic algorithms are synchronous, this could give us a basis in the future to compare the Pool GA with other existing parallel genetic algorithms.

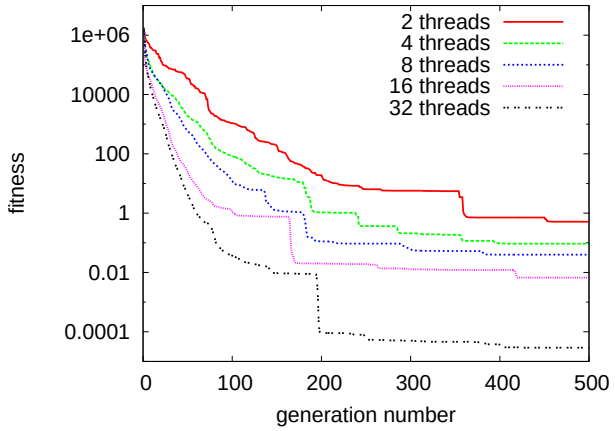

Fig. 2. Benchmark function f_1: Synchronous operation, average speed of convergence over 500 generations with population size 16 per thread

We have used the benchmark functions for these simulations. Figs. 2, 3, 4, and 5 show the results for function f_1, f_2, f_3 and f_4 respectively. The plots show the average of the best fitness value seen in each generation by each thread under varying number of threads.

In all the remaining sections of this chapter, the results provided are for asynchronous

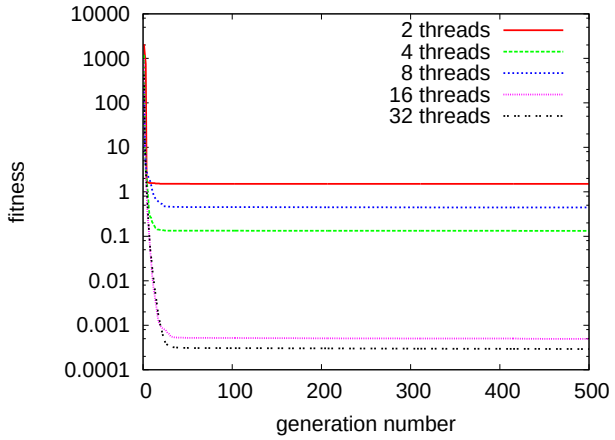

Fig. 3. Benchmark function f_2: Synchronous operation, average speed of convergence over 500 generations with population size 16 per thread

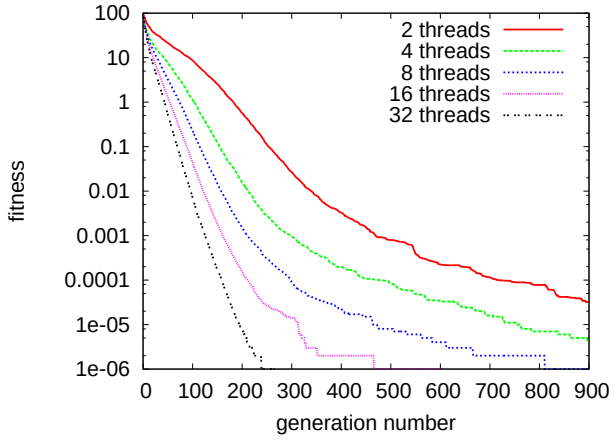

Fig. 4. Benchmark function f_3: Synchronous operation, average speed of convergence over 900 generations with population size 50 per thread

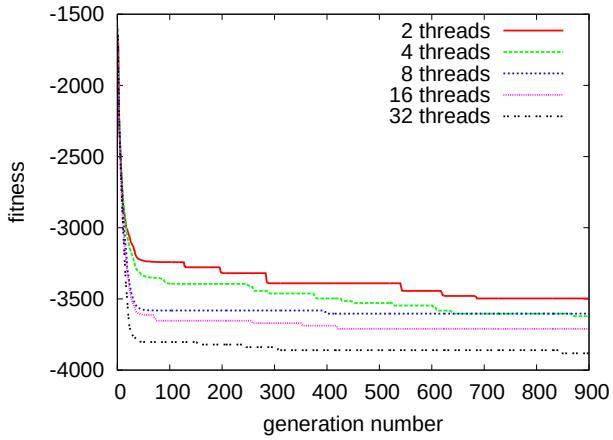

Fig. 5. Benchmark function f_4: Synchronous operation, average speed of convergence over 900 generations with population size 50 per thread

operation.

C. Performance on Benchmark Functions for Asynchronous Operation

We now provide simulation results for the Pool GA applied to the benchmark functions studied in [3] when the participating threads behave asynchronously. The plots show the average of the best fitness value seen in each generation by each thread under varying number of threads. Figs. 6, 7, 8, and 9 show the results.

On all four functions, the common behavior observed is that the more threads, the faster the convergence to a solution with better fitness. For f_1, f_2 and f_3 which have optimum value zero, the Pool GA reaches quite close to the optimum value. The function f_4 has optimal value -4189 and it is considered quite hard to reach [3]. We see in Fig. 9 that with greater number of threads a better value for average of the best fitness seen by each thread per generation is reached. For a different perspective on the computation of f_4, in Fig. 10 we plot the best value seen among all the threads at a particular generation instead of the average of the best value seen by all the threads. This gives a different look

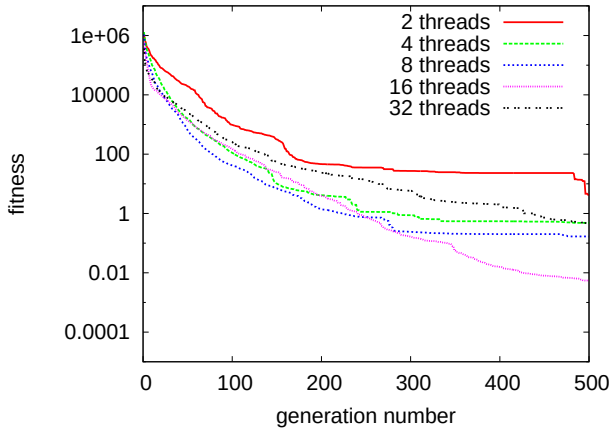

Fig. 6. Benchmark function f_1: Average speed of convergence over 500 generations with population size 16 per thread

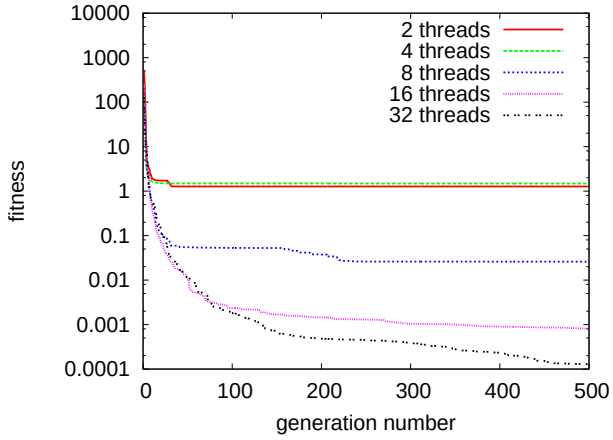

Fig. 7. Benchmark function f_2: Average speed of convergence over 500 generations with population size 16 per thread

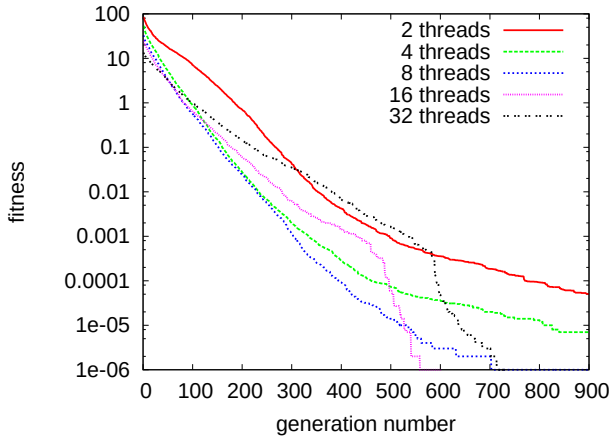

Fig. 8. Benchmark function f_3: Average speed of convergence over 900 generations with population size 50 per thread

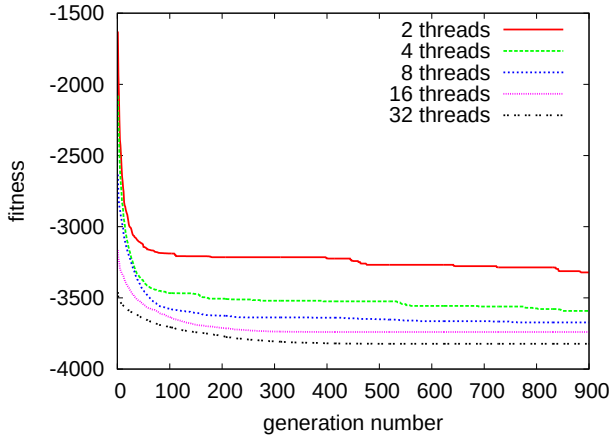

Fig. 9. Benchmark function f_4: Average speed of convergence over 900 generations with population size 50 per thread

on the progress of the GA. It appears finding a good solution for f_4 is easy, but finding an excellent one is hard.

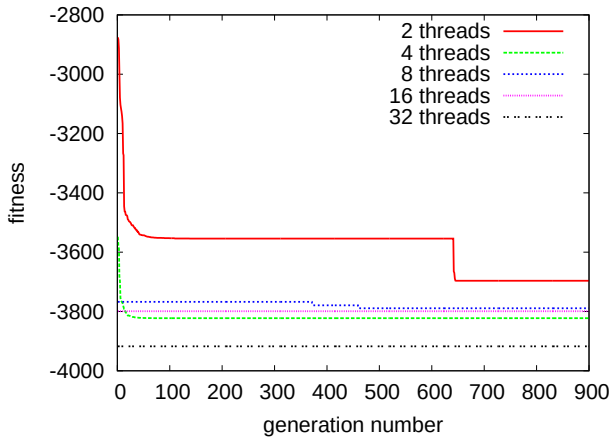

Fig. 10. Benchmark function f_4: Speed of convergence over 900 generations with population size 50 per thread

On close observation of the results of Figs. 6 and 8, we see that for the functions f_1 and f_3 the 32 thread case is an out-lier to the general trend observed. This is because the metric we use to show the progress of the GA is the average of the best fitness value seen in each generation by each thread. Thus each point on the graph corresponding to a particular generation number, say n, is the average of the best value seen by each of the participating threads in generation n. Two aspects of such a plot must be made clear. Firstly because the execution is asynchronous, the time when one thread executes generation x may be much earlier or later than when another thread executes generation x. For instance for the case of 8 threads, thread 1 may execute generation 5 at time t, thread 2 may execute generation 5 at time $t + 10$ while thread 3 may execute generation 5 at time $t - 5$. Thus when we average the best values for generation 5 we are not averaging values that were obtained at the same real-times. Secondly, in spite of the above real-time anomaly, these plots are

still good indicators of the progress of the GA. To illustrate this, continuing the above example, say thread 1 executes generation 6 at time $t + 3$, thread 2 executes generation 6 at time $t + 15$ while thread 3 executes generation 6 at time $t - 1$. Thus the data we use to find the average of generation 6 are generated at later times than the values used for the average of generation 5. Getting back to our 32 thread out-lier case, we note that for a large number of threads like 32 in any generation, some threads have access to an excellent individual while some do not, thus making the average value of fitness seem bad. If we look at only the best individual found, which would be the actual result of the GA, the 32 threads simulation actually obtains the optimum value of zero. Moreover due to the asynchrony some thread in the simulation may see an individual with the best fitness as early as generation 1. Tables II and III reflects this fact; they provide the best value of fitness seen for each number of threads and the generation number when any thread in the simulation first saw an individual with that fitness.

Table II. Benchmark function f_1: Best fitness and first generation when the best fitness was seen

Number of Threads	Best Fitness	First Generation
2	0.009116	486
4	0.004045	303
8	0.004701	120
16	0.0	1
32	0.0	1

In Figs. 9 and 10 we observe that the simulation never achieves the optimal value of fitness, i.e., -4189. We believe that part of the difficulty that our Pool GA had with

Table III. Benchmark function f_3: Best fitness and first generation when the best fitness was seen

Number of Threads	Best Fitness	First Generation
2	0.000041	868
4	0.000001	798
8	0.0	387
16	0.0	1
32	0.0	1

finding optimal solutions to f_4 is due to the simplistic nature of the Selection, Mutation and Crossover operators used in our simulation. We conjecture with better operators tuned to the specific function the results will improve.

D. Performance on Product Lifecycle Design Problem for Asynchronous Operation

We now provide results for our Pool GA applied to the Product Lifecycle Design problem. Figs. 11 and 12 show the results for two different target customer groups. Plots show the best fitness value seen by the simulation in each generation for varying number of processors. As can be seen, using fewer threads it takes more generations to converge to the optimal fitness values of 0.83 and 0.63 respectively, as compared to using 8 or 32 threads. We anticipate this difference will be more and more pronounced as the problem being solved becomes larger and more complex.

Currently the Lifecycle Design problem does not appear particularly difficult to solve. Note that simply choosing around 3000 candidate solutions at random and finding the one with the best fitness appears to work quite well, without the need to do any additional

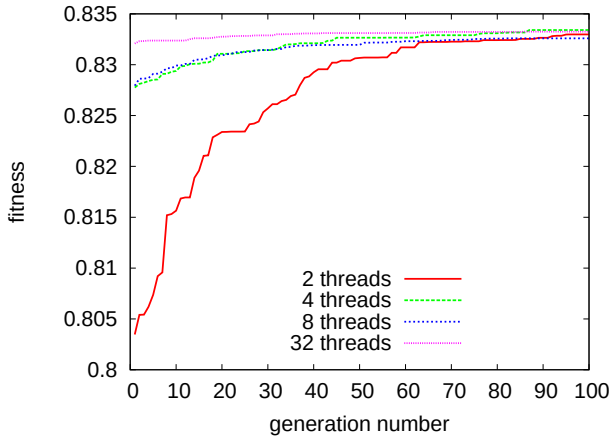

Fig. 11. Lifecycle Design problem for neutral customer group: Speed of convergence over 100 generations with population size 50 per thread

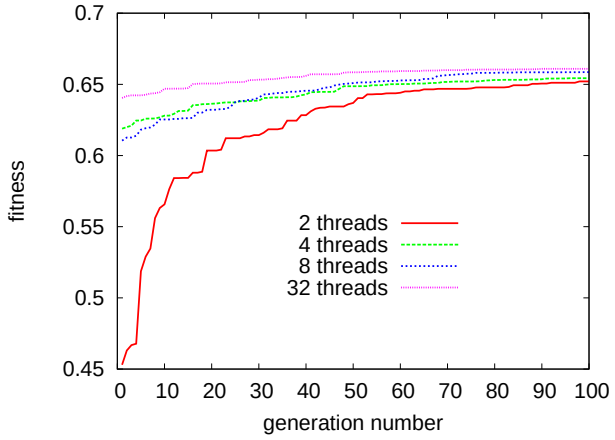

Fig. 12. Lifecycle Design problem for technophile customer group: Speed of convergence over 100 generations with population size 50 per thread

computation.

However for our simulations we have used a simple version of the problem which focuses on one customer group and optimizes only a single objective instead of multiple objectives. The development of this problem is still a work in progress and we anticipate in the future that the problem will become essentially so large and complex that using a distributed genetic algorithm will pay dividends.

E. Fault-Tolerance to Crash Failures

We performed simulations to test the fault-tolerance of our Pool GA. We simulated crash failures of processors by ending each thread at the beginning of each of its generations with probability $\frac{1}{2g}$, where g is the number of generations in the run. Thus, over the course of the run, we expect at most half the threads to crash. The simulations of Figs. 7 and 8 were repeated under this fault model and the results are shown in Figs. 13 and 14. We see that the convergence rate is not greatly affected, even though, on an average, half of the participating processors crash.

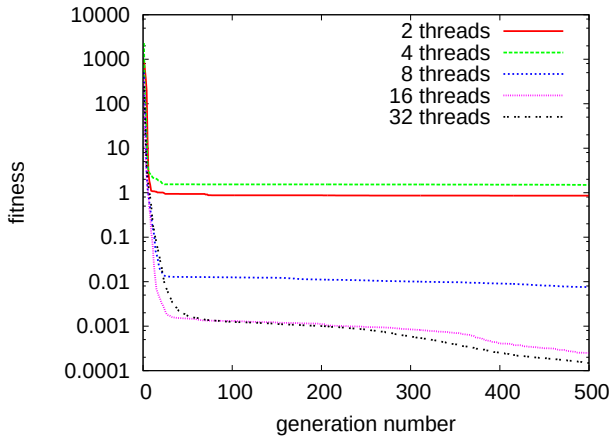

Fig. 13. Benchmark function f_2 with crashes: Average speed of convergence over 500 generations with population size 16/thread, failure probability 1/1000

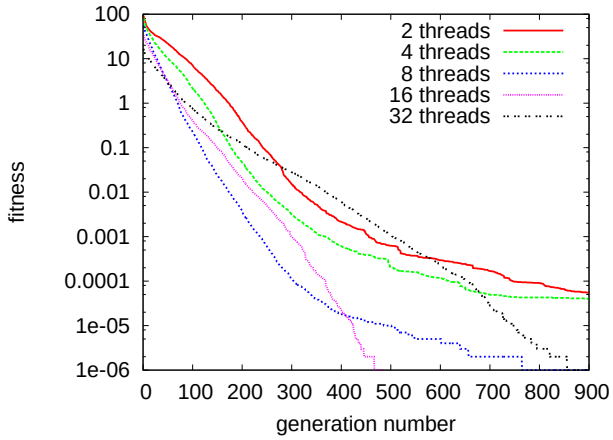

Fig. 14. Benchmark function f_3 with crashes: Average speed of convergence over 900 generations with population size 50/thread, failure probability 1/1800

F. Fault-Tolerance to Byzantine Failures

Recall that we model Byzantine behavior of processors by the Anti-Elitism characteristic where a Byzantine faulty processor writes back newly generated individuals into the pool only if the individual it is trying to replace from the pool is better. In our simulations, when we say $f\%$ of processors are Byzantine in a total of N threads, then $\lfloor f * N/100 \rfloor$ processors are Byzantine. For instance when we say, for a simulation with 2 threads, 80% of the processors are Byzantine, $\lfloor 80 * 2/100 \rfloor = 1$ processor is Byzantine. The results plotted are from data generated by only the correct processors in the simulation; the output of the Byzantine faulty processors are ignored.

Our first set of plots show how the Pool GA performs as the percentage of Byzantine processors in the system increases. We provide the results when 33%, 60% and 80% of the processors are Byzantine. Figs. 15, 16 and 17 show the results for function f_1, while Figs. 18, 19 and 20 show the results for function f_3. We observe the fault-tolerance of the Pool GA even when faced with this malignant kind of failure. The final fitness values

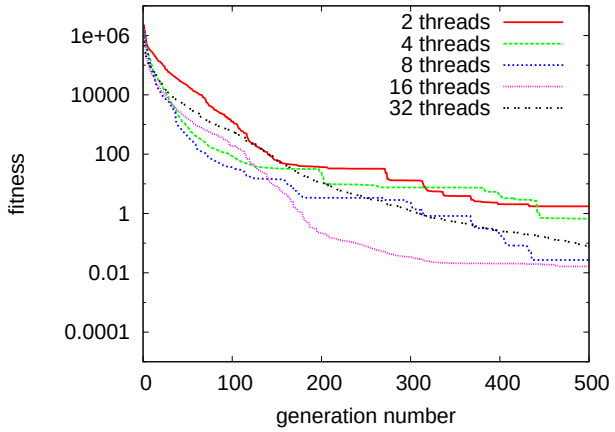

Fig. 15. Benchmark function f_1 with 33% Byzantine faults: Average speed of convergence over 500 generations with population size 16/thread

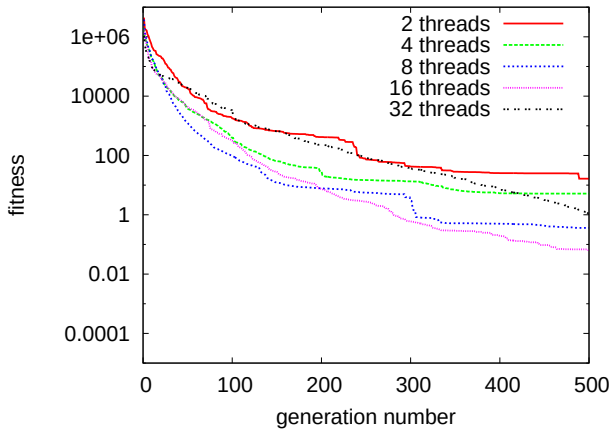

Fig. 16. Benchmark function f_1 with 60% Byzantine faults: Average speed of convergence over 500 generations with population size 16/thread

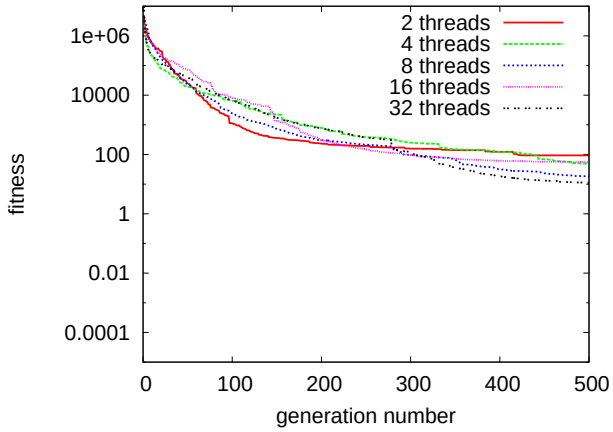

Fig. 17. Benchmark function f_1 with 80% Byzantine faults: Average speed of convergence over 500 generations with population size 16/thread

achieved in the 33% and 60% cases are not very different from those achieved in the non-faulty cases. The performance is worse for the 80% case, yet the GA still makes significant progress in the right direction. We observe a similar trend for both f_1 and f_3: the larger the number of correct threads, the better the convergence. This makes a strong case for using increased levels of distribution in solving GA problems.

The percentage of faulty processors has a pronounced effect on the convergence of the fitness values. This can be seen in Figs. 21 and 22 which compare the performance for 8 threads with varying Byzantine failure percentages for functions f_1 and f_3 respectively.

G. Distribution of Fitness of Individuals in the Pool

In previous sections we have mostly looked at the average of the best values seen by the processors involved in the Pool GA in each generation. We have seen that the Pool GA has good fault-tolerance. For crash failures, the average best values (Figs. 13 and 14) obtained are almost as good as the values obtained for the corresponding cases with no failure (Figs. 7 and 8). For the Byzantine failure case, when 33% of the processors in the

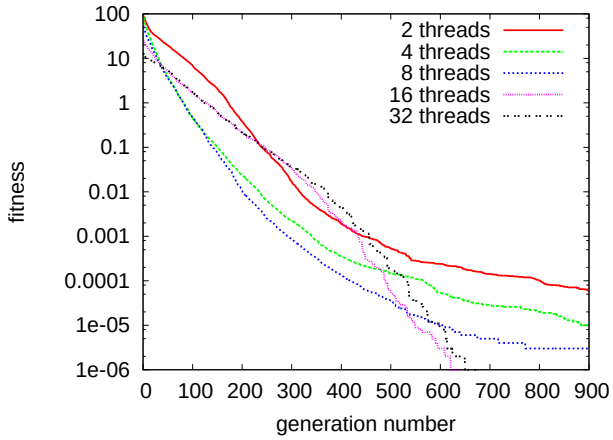

Fig. 18. Benchmark function f_3 with 33% Byzantine faults: Average speed of convergence over 900 generations with population size 50/thread

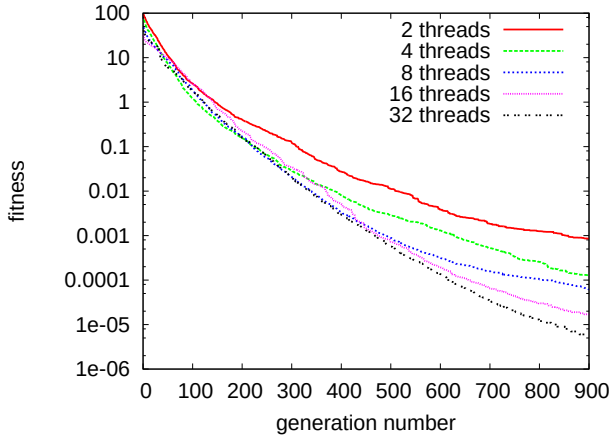

Fig. 19. Benchmark function f_3 with 60% Byzantine faults: Average speed of convergence over 900 generations with population size 50/thread

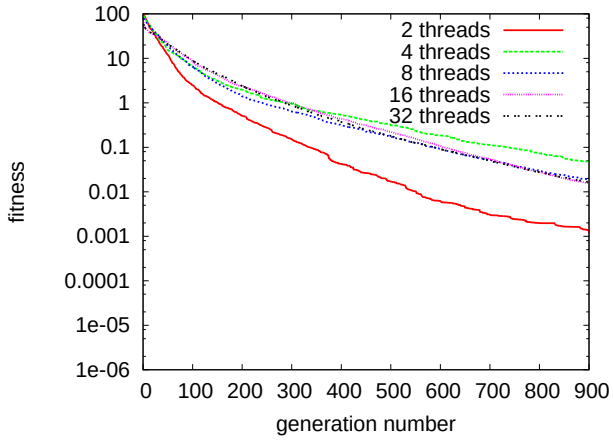

Fig. 20. Benchmark function f_3 with 80% Byzantine faults: Average speed of convergence over 900 generations with population size 50/thread

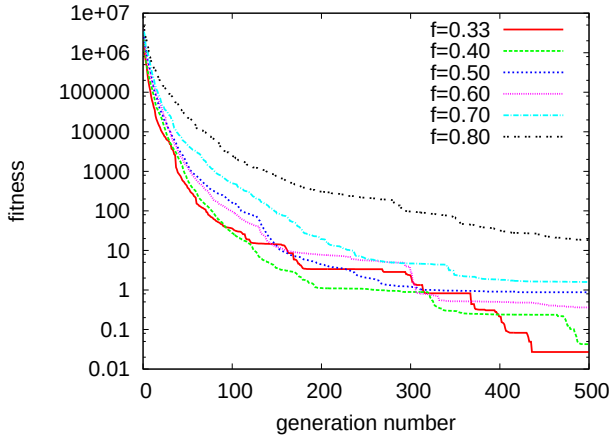

Fig. 21. Benchmark function f_1 with 8 threads and varying percentage of Byzantine faults: Speed of convergence over 500 generations with population size 16/thread

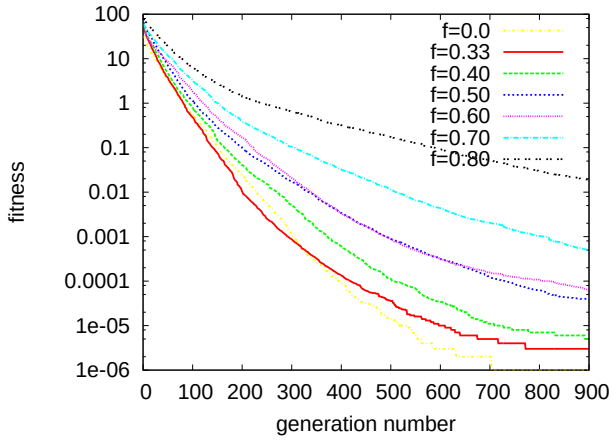

Fig. 22. Benchmark function f_3 with 8 threads and varying percentage of Byzantine faults: Average speed of convergence over 900 generations with population size 50/thread

system are Byzantine (Figs. 15 and 18) the average of the best values is still comparable to the no-failure case (Figs. 6 and 8). For the the 60% Byzantine processors case (Figs. 16 and 19) the results are still good but the average fitness values worsen about 10 times. For the 80% Byzantine processors case, the results deteriorate (Figs. 17 and 20) and there is an order-of-magnitude difference in the average of the best values as compared to previous plots.

Looking at the average of the best values seen by all the threads is a good indicator of the performance of the GA; however, there are some interesting aspects that are missed out. Firstly, the result of the GA is the absolute best value seen and that value could be much less than the average of the best values seen by each thread. Secondly, it is interesting to see what the fitness values of the various individuals in the pool are: are all individuals in the pool mere replicas of the best individual, are most individuals in the pool similar to the best individual or are most individuals of poor fitness? How does this vary with crash and Byzantine failures? To answer these questions we look at the distribution of the fitness of the individuals in the pool.

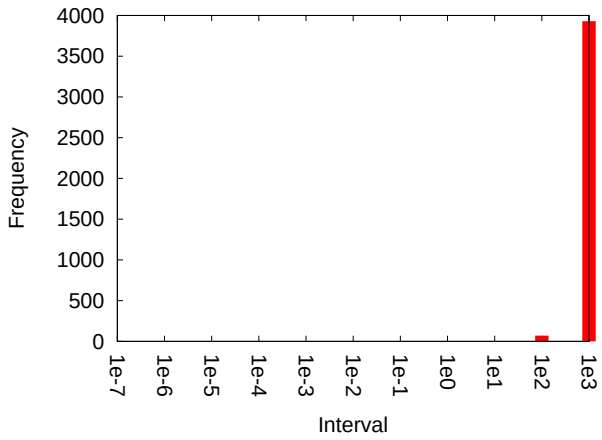

Fig. 23. Distribution of fitness of individuals in initial pool for function f_3 with 8 threads

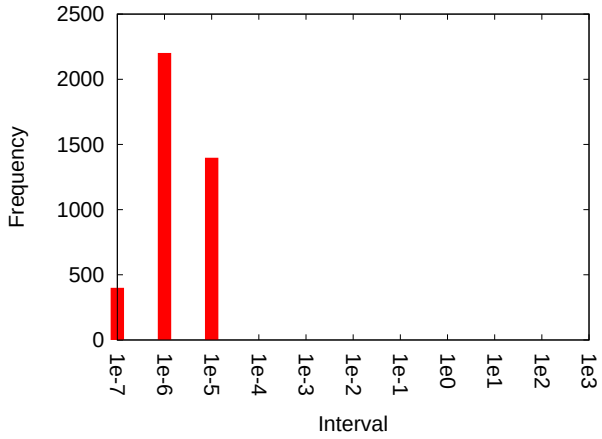

Fig. 24. Distribution of fitness of individuals in final pool for function f_3 with 8 threads under no failures

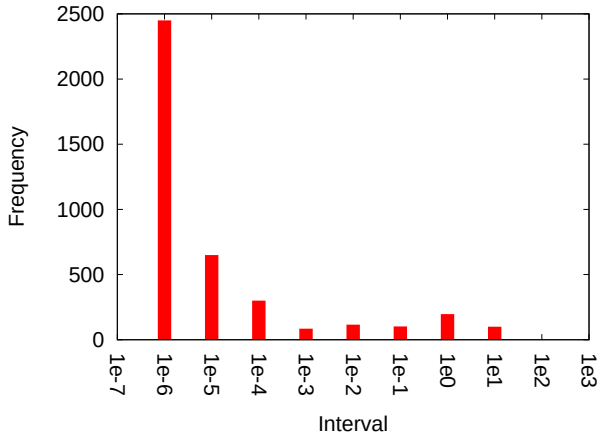

Fig. 25. Distribution of fitness of individuals in final pool for function f_3 with 8 threads under crash failures (1/1800 probability of crash in each generation)

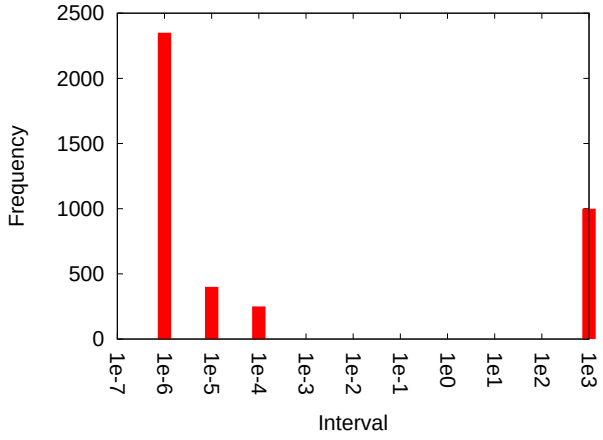

Fig. 26. Distribution of fitness of individuals in final pool for function f_3 with 8 threads under 33% Byzantine failures

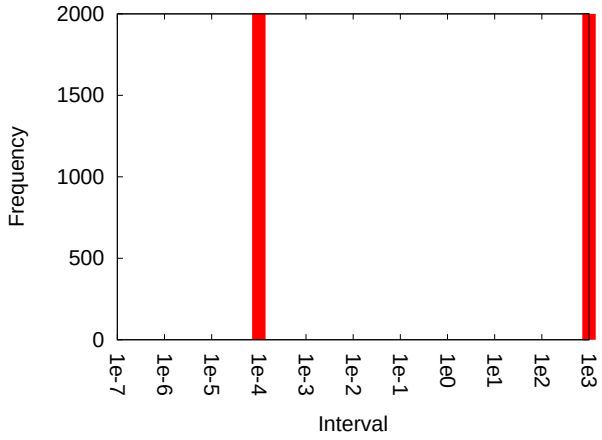

Fig. 27. Distribution of fitness of individuals in final pool for function f_3 with 8 threads under 60% Byzantine failures

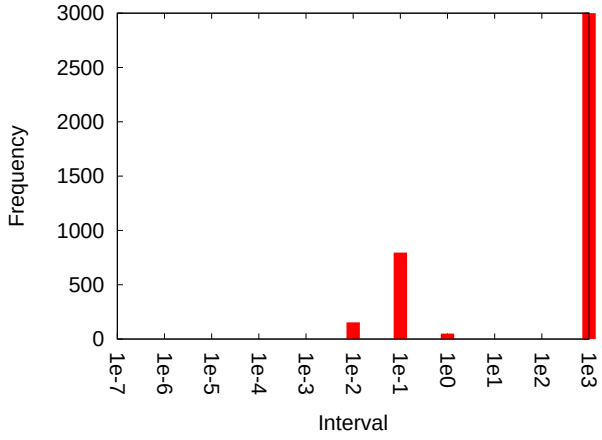

Fig. 28. Distribution of fitness of individuals in final pool for function f_3 with 8 threads under 80% Byzantine failures

In this section we look at the distribution of the fitness of the individuals in the pool. We use the function f_3 and run the Pool GA with 8 threads. Simulation parameters are kept the same as those for previous simulations of f_3, i.e., a population of 50/thread, total 900 generations, same probability of mutation, selection, crossover. For the crash failure simulation we use the same probability of failure, $1/1800$, as before. The initial pool is kept the same for each of the simulations and the distribution of the initial pool is provided in Fig. 23. We look at the distribution of the pool at the end of the simulation of the no-failure case (Fig. 24), crash failure case (Fig. 25) and Byzantine failures with 33%, 60% and 80% of the processors being Byzantine (Figs. 26, 27 and 28). Each of the distributions is plotted with a logarithmic x-axis starting from 10^{-7} and ending with 10^3. The bar corresponding to 10^{-7} gives the number of individuals whose fitness f lies in $10^{-8} < f \leq 10^{-7}$. The bar corresponding to 10^{-6} gives the number of individuals whose fitness f lies in $10^{-7} < f \leq 10^{-6}$ and so on. We did not see any individuals with fitness less than 10^{-8} for the above simulations so the range of our axes is appropriate. Each plot contains data of 10 runs. Thus the sum of the y-values of the bars is the sum of the pool sizes of all the runs. In our case the population is 50/thread and hence the pool size for one run is $50 * 8 = 400$. For the 10 runs we observe a total of 4000 individuals and that is the number of individuals in each distribution.

Fig. 23 shows that initial fitness values of the individuals in the pool is quite poor. For the simulation with no failures (Fig. 24) we observe that in the final pool most individuals have close to optimum fitness; the quality of the individuals in the pool is thus overall very good. For the crash failure simulation (Fig. 25), we see a greater spread in the distribution; however, a majority segment of the pool still has very good fitness values. For the Byzantine failure simulations (Figs. 26, 27 and 28) we observe that the pool appears to have two partitions. One partition has individuals of good (low) fitness values while the other partition has bad (high) fitness values. As the percentage of Byzantine failures increases, the

number of individuals in the bad fitness partition keeps increasing. This general trend is to be expected because with our Anti-Elitism approximation of Byzantine behavior, the correct processors try to reduce the fitness value of individuals in the pool while the Byzantine processors try to increase the fitness value of the individuals in the pool. It is an interesting open problem, however, to study why our Anti-Elitism approximation of Byzantine failure leads to the bimodal distribution observed.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

In this thesis we proposed a new architecture for distributed genetic algorithms in which the participating processors interact in an asynchronous, loosely coupled manner through shared objects. This architecture is tailored to take advantage of the state of the art in distributed computing by allowing processors with different speeds to cooperatively solve a problem. The architecture also provides fault-tolerance to processor failures by allowing the data to be decoupled from the processors. Fault-tolerance is a crucial property in today's world where the availability of large numbers of processors increases the chance that some of the processors will fail.

In the future, we would like to explore the pool model further to study optimum parameters for convergence such as the relation between choices of pool size, processor population size, and the effect of the strategy for writing back to the pool. Currently the pool of individuals is a passive store of data; we would like to explore the possibility of making the pool more intelligent; for instance, can the pool automatically replicate individuals of greater fitness? We would also like to provide an implementation of the Pool GA on a parallel programming framework like OpenMP or MPI and test with the full version of the Lifecycle Design problem. In terms of parallel implementations it will be interesting to see whether the pool architecture fits in well with Google's Mapreduce paradigm [27], which would make the parallel programming easier. From a distributed shared memory perspective, we would like to define the semantics of the pool as a linearizable shared memory data structure [24]. Finally we would also like to explore different ways of modelling Byzantine failure of processors for our Pool GA.

REFERENCES

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press, 1992. [Online]. Available: http://portal.acm.org/citation.cfm?id=129194

[2] D. Whitley, “A genetic algorithm tutorial,” *Stat. and Comp.*, vol. 4, no. 2, pp. 65–85, June 1994. [Online]. Available: http://dx.doi.org/10.1007/BF00175354

[3] R. Salomon, “Reevaluating genetic algorithm performance under coordinate rotation of benchmark functions; a survey of some theoretical and practical aspects of genetic algorithms,” *BioSystems*, vol. 39, pp. 263–278, 1995.

[4] D. Mangun and D. Thurston, “Incorporating component reuse, remanufacture, and recycle into product portfolio design,” *IEEE Trans. Eng. Manag.*, vol. 49, no. 4, pp. 479–490, Nov 2002.

[5] L. Lamport, R. Shostak, and M. Pease, “The Byzantine generals problem,” *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, 1982.

[6] G. Roy, H. Lee, J. L. Welch, Y. Zhao, V. Pandey, and D. Thurston, “A distributed pool architecture for genetic algorithms,” in *IEEE Congress on Evolutionary Computation*, May 2009, pp. 1177–1184.

[7] J. E. Rowe, “Genetic algorithm theory,” in *Proc. of the GECCO Conf. Companion on Genetic and Evolutionary Computation*. New York, NY: ACM, 2008, pp. 2535–2558.

[8] E. Cantú-Paz, “A survey of parallel genetic algorithms,” Illinois Genetic Algorithms Laboratory, University of Illinois, Tech. Rep. 97003, 1997.

[9] K. Deb, P. Zope, and A. Jain, “Distributed computing of Pareto-optimal solutions with evolutionary algorithms,” in *Evolutionary Multi-Criterion Optimization*, ser. Lecture Notes in Computer Science, vol. 2632. Springer, 2003. [Online]. Available: http://www.springerlink.com/content/xm2qxu4hbrkm6jk5/

[10] N. Carriero and D. Gelernter, “Linda in context,” *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[11] N. Carriero, D. Gelernter, and J. Leichter, “Distributed data structures in Linda,” in *Proc. of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL ’86*. New York, NY: ACM, 1986, pp. 236–242.

[12] G. Sutcliffe and J. Pinakis, “Prolog-D-Linda: An embedding of Linda in SICStus Prolog,” in *Proc. of the Joint Workshop on Distributed and Parallel Implementation of Logic Programming Systems*, 1992, pp. 70–79.

[13] M. Davis, L. Liu, and J. Elias, “VLSI circuit synthesis using a parallel genetic algorithm,” in *Proc. of the 1st IEEE Conf. on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, vol. 1, June 1994, pp. 104–109.

[14] J. C. C. Litrán, X. Defago, and K. Satou, “Asynchronous peer-to-peer communication for failure resilient distributed genetic algorithms,” in *Proc. of the 15th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS)*, 2003, pp. 769–773.

[15] K. Li and P. Hudak, “Memory coherence in shared virtual memory systems,” *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[16] D. K. Gifford, “Weighted voting for replicated data,” in *Proc. of the Seventh ACM Symposium on Operating Systems Principles, SOSP ’79*. New York, NY: ACM,

1979, pp. 150–162.

[17] V. Pandey, D. Thurston, K. Kanjani, and J. Welch, “Distributed data sources for life-cycle design,” in *Proc. of the 16th Int. Conf. on Engineering Design (ICED)*, 2007.

[18] J. I. Hidalgo, J. Lanchares, F. Fernández de Vega, and D. Lombrańa, “Is the island model fault tolerant?” in *Proc. of the 2007 GECCO Conf. Companion on Genetic and Evolutionary Computation*. New York, NY: ACM, 2007, pp. 2737–2744.

[19] D. L. Gonzalez and F. F. de Vega, “On the intrinsic fault-tolerance nature of parallel genetic programming,” in *Proc. of 15th EUROMICRO Int. Conf. on Parallel, Distributed and Network-Based Processing, PDP ’07*, Feb. 2007, pp. 450–458.

[20] J. J. Merelo, A. M. García, J. L. J. Laredo, J. Lupión, and F. Tricas, “Browser-based distributed evolutionary computation: Performance and scaling behavior,” in *Proc. of the 2007 GECCO Conf. Companion on Genetic and Evolutionary Computation*. New York, NY: ACM, 2007, pp. 2851–2858.

[21] F. Kimura, “A computer-supported approach to life cycle design of eco-products,” in *Proc. of 5th Int. Conference on EcoBalance*, 2002, pp. 451–452.

[22] V. Pandey and D. Thurston, “Non-dominated strategies for decision based design for product reuse,” in *Proc. of ASME Int. Design Engineering Technical Conferences*, 2007.

[23] N. Sakai, G. Tanaka, and Y. Shimomura, “Product life cycle design based on product life control,” in *3rd Int. Symposium on Environmentally Conscious Design and Inverse Manufacturing, EcoDesign ’03*, Dec. 2003, pp. 102–108.

[24] M. P. Herlihy and J. M. Wing, “Linearizability: a correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 463–492, 1990.

[25] D. P. Anderson, “Boinc: A system for public-resource computing and storage,” in *Proc. of the 5th IEEE/ACM Int. Workshop on Grid Computing GRID ’04*, 2004, pp. 4–10.

[26] K. Deb, “Kangal codes.” [Online]. Available: http://www.iitk.ac.in/kangal/codes.shtml

[27] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *OSDI*, 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

APPENDIX A

PRODUCT LIFECYCLE DESIGN APPLICATION

We consider design of a product portfolio to cover different customer market segments, over multiple lifecycles. Four market segments are defined: technophile, utilitarian, greens and neutral in terms of their relative preference for performance, cost, and environmental impact. Manufacturers need to make optimal design decisions to maximize the total product portfolio utility, which is a function of cost, environmental impact and performance. In each market segment, customers have their own preferences and willingness to make tradeoffs, which together define their utility functions.

The decision variables are the discrete design decisions for each component of the product in each lifecycle. The resulting optimization problem is large; for example, for five lifecycles of a single product comprising 12 components, about 10^{36} solutions are possible if each component can be reused, remanufactured, recycled or replaced. Exhaustive enumeration of all solutions is not feasible. Consideration of multiple products per lifecycle (product portfolio) will undoubtedly increase the problem complexity even further.

Max U_p s.t.

$$
\begin{aligned}
U_p &= \frac{1}{K_p}\left[\prod_{a\in\{C,E,R\}}(K_p k_{p,a} U_{p,a}+1)-1\right] \\
U_{p,C} &= \frac{C_{p,\max}-C_p}{C_{p,\max}-C_{p,\min}} \\
U_{p,E} &= \frac{E_{p,\max}-E_p}{E_{p,\max}-E_{p,\min}} \\
U_{p,R} &= \frac{R_p-R_{p,\min}}{R_{p,\max}-R_{p,\min}} \\
C_p &= \sum_{l=1}^{L_p}(C_{p,l}+Q_{p,l}) \\
C_{p,l} &= \sum_{i=1}^{s}\left[x_{p,l,i,1}\left(C_{p,l,i,8}+\sum_{n=1}^{5}C_{p,l,i,n}\right)\right. \\
&\quad +x_{p,l,i,2}\left(C_{p,l,i,4}\right)+x_{p,l,i,3}\sum_{n=3}^{6}C_{p,l,i,n} \\
&\quad \left.+x_{p,l,i,4}\left(C_{p,l,i,7}+\sum_{n=2}^{5}C_{p,l,i,n}\right)\right] \\
E_p &= \sum_{l=1}^{L_p}E_{p,l} \\
E_{p,l} &= \sum_{i=1}^{s}\left[x_{p,l,i,1}\left(E_{p,l,i,8}+\sum_{n=1}^{5}E_{p,l,i,n}\right)\right. \\
&\quad +x_{p,l,i,2}(E_{p,l,i,4})+x_{p,l,i,3}\sum_{n=3}^{6}E_{p,l,i,n} \\
&\quad \left.+x_{p,l,i,4}\left(E_{p,l,i,7}+\sum_{n=2}^{5}E_{p,l,i,n}\right)\right] \\
R_p &= \min\{R_{p,l}: l=1,\ldots,L_p\} \\
R_{p,l} &= f(R_{p,l,1},\ldots,R_{p,l,s}) \\
R_{p,l,i} &= \exp\left(-\left[\frac{t_{p,l,i}}{\Theta_i}\right]^{b_i}\right) \\
t_{p,l,i} &= g(t_{p,l-1,i},x_{p,l,i,1},...,x_{p,l,i,4})+a_{p,l}, l>0 \\
t_{p,0,i} &= 0 \\
\sum_{l=1}^{L_p}a_{p,l} &= 10
\end{aligned}
$$

$C_{p,\min} \leq C_p \leq C_{p,\max}$

$E_{p,\min} \leq E_p \leq E_{p,\max}$

$R_{p,\min} \leq R_p \leq R_{p,\max}$

$x_{p,l,i,1}, x_{p,l,i,2}, x_{p,l,i,3}, x_{p,l,i,4} \in \{0, 1\}$

$x_{p,l,i,1} + x_{p,l,i,2} + x_{p,l,i,3} + x_{p,l,i,4} = 1$

Notes:

- p = index of specific product in the product portfolio
- $K_p =$ normalizing constant for product p
- a ranges over attributes C (cost), E (environmental impact), and R (reliability)
- $k_{p,a} =$ scaling constant corresponding to attribute a for product p
- $U_{p,a} =$ utility of attribute a for product p
- $C_p =$ total cost of product p; between $C_{p,\min}$ and $C_{p,\max}$
- $E_p =$ total environmental impact of product p; between $E_{p,\min}$ and $E_{p,\max}$
- $R_p =$ the minimum reliability in all lifecycles of product p; between $R_{p,\min}$ and $R_{p,\max}$
- $L_p =$ number of lifecycles of product p
- $C_{p,l} =$ cost associated with product p in lifecycle l
- $Q_{p,l} =$ profit margin of product p in lifecycle l
- $s =$ number of components in the product (all products in portfolio have same number of components)

- $x_{p,l,i,z}$ = binary design decision for component i of product p during lifecycle l; z ranges from 1 to 4 with 1 indicating reuse, 2 remanufacturing, 3 recycling, and 4 new; for a fixed p, l, and i, exactly one of the four variables should be true
- $C_{p,l,i,n}$ = cost of operation n for component i of product p in lifecycle l; n ranges from 1 to 8 with 1 indicating new material acquisition, 2 manufacturing/forming, 3 assembly, 4 take-back, 5 disassembly, 6 remanufacturing, 7 recycling, and 8 disposal
- $E_{p,l}$ = environmental impact associated with product p in lifecycle l
- $E_{p,l,i,n}$ = environmental impact of operation n for component i of product p in lifecycle l
- $R_{p,l}$ = reliability of product p in lifecycle l; based on component reliabilities and failure model assumed
- $R_{p,l,i}$ = reliability of component i of product p in lifecycle l
- f = function modeling failure mode for the product
- Θ_i = characteristic life of component i
- b_i = slope of Weibull reliability curve for component i
- $t_{p,l,i}$ = age of component i in product p at end of lifecycle l
- g = function modeling how the design decisions ($x_{p,l,i,z}, z = 1, \ldots, 4$) impact component i's end of lifecycle age
- $a_{p,l}$ = product p's usage time in lifecycle l

VITA

Gautam Samarendra N Roy obtained a Bachelor of Technology in electronics and communication engineering from Indian Institute of Technology Guwahati in 2005. Prior to joining Texas A&M in 2007, he spent two years working as a software engineer in the System LSI Division of Samsung, India. At Texas A&M, he worked in the Distributed Computing Group with Dr. Jennifer Welch and graduated with his master's in Computer Engineering in December 2009.

He can be reached at c/o: Dr. Jennifer Welch, Department of Computer Science and Engineering, Texas A&M University, TAMU 3112, College Station, TX - 77843-3112. His email is groys@tamu.edu