DESIGN OF A PARALLEL GENETIC ALGORITHM FOR THE INTERNET

D. Joseph and W. Kinsner
Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada, R3T-5V6
e-mail: kinsner@ee.umanitoba.ca

Abstract

This paper proposes that a parallel implementation of the genetic algorithm (GA) on the Internet will improve the algorithm's performance. It is motivated by the possibility of aiding research into complex search and optimization problems that use the GA. Requirements and constraints regarding parallelization of the GA are identified. A parallel GA is developed for an ideal PRAM architecture and is shown to have an asymptotic running time of *O(log n)*, an improvement over the sequential GA. A parallel GA is also designed for a Unix network and has an asymptotic running time comparable to the ideal system. The algorithm is a decentralized, asynchronous, and fault-tolerant design that matches characteristics of the network. The GA population is divided into colonies that are distributed among processors. Trade policies are executed for the exchange of genes.

I INTRODUCTION

Many problems studied in research and development require a parameter search or optimization of some kind. Exhaustive examination of the search space is often impractical. A feasible approach, however, is to use an inexhaustive search technique that yields suboptimal but reasonable results in a realistic time frame. One such technique is the *genetic algorithm* (GA). While the GA may offer results in realistic time frames, it may still require days or weeks of execution time when tackling difficult and not-well-understood problems. It is thus desirable to develop a faster implementation of the GA to improve the pace of related research.

Most scientific and engineering research institutions, such as universities, possess networks of computers, often running the Unix operating system and communicating with Internet protocols. Such an array of computers represents a significant, and readily available, parallel processing power. This paper proposes that these computers may be used cooperatively to improve the performance of the GA.

A design of the GA for parallel execution on a network of computers should meet certain criteria. Table 1 highlights these requirements.

Table 1. Requirements of a parallel GA.

(1) Improves speed and search power of GA
(2) Does not undermine GA fundamentals
(3) Minimal communication/synchronization
(4) Does not compromise security
(5) Is considerate to other users
(6) Is adaptable to network dynamics
(7) Launches quickly from any machine
(8) Can function for long periods of time
(9) Possesses fault tolerance
(10) Is capable of termination

It might be argued that improving speed and search power, as stated in Requirement (1), are really the same goal. That is, a parallel implementation may perform the same execution in a shorter time or more execution in the same time as compared to a non-parallel system. While this trade-off usually exists, there are other possibilities. For example, the limitations of a sequential system may prevent it from performing a task at all, while a parallel system may complete it.

Requirement (2) stipulates that a parallel GA must adhere to the fundamental principles of the algorithm. Running the GA independently on every machine in the network would be unacceptable as a parallel solution because it would violate principles of selection and reproduction in the GA.

II THEORETICAL CONSIDERATIONS

2.1 Asymptotic Running Times

To analyse the algorithms developed in this paper, we compared their *asymptotic running times*. This technique simplifies analysis by ignoring details such as machine speeds and compilers. It measures the fundamental behaviour of the algorithm with respect to the input size. Because of varying notation and definitions in the literature, we shall use the following notation and definitions [CoLR89]. If $g(n)$ is an upper bound of $f(n)$ and $h(n)$ is its lower bound, then

$f(n) = O(g(n))$ if there exists positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$. (1)

$f(n) = \Omega(h(n))$ if there exists positive constants c and n_0 such that $0 \le c \cdot h(n) \le f(n)$ for all $n \ge n_0$. (2)

2.2 Flynn's Taxonomy

Parallel processing can be realized in many different computing architectures. Flynn's taxonomy classifies these architectures based on the pattern of interaction between instructions and data [Kins96]. A *multiple instruction multiple data* (MIMD) architecture consists of processing elements (PEs) that can execute different instructions on different data in parallel. A MIMD *multicomputer* system is where communication between PEs is via message passing (as opposed to shared memory). This classification represents the most loosely coupled architecture in Flynn's taxonomy.

Because of the relative independence of PEs, MIMD multicomputer systems are difficult to use. However, these systems are also more likely to support fault-tolerance for the same reason. A fault that occurs in one PE is less likely to hinder other PEs. Thus, a good parallel design will take advantage of hardware robustness and continue despite partial shutdown of some PEs. In addition, a MIMD multicomputer architecture may be the parallel architecture that is most readily available. Any subset of the Internet, with two or more computers, is an example of a MIMD multicomputer architecture.

2.3 Parallelization Granularity

The *granularity* of a parallel algorithm summarizes how the overall task is divided into parallel subtasks [Posk97]. *Fine-grain* parallelization sees the overall task as a set of machine instructions where subtasks are a few machine instructions long. *Coarse-grain* parallelization sees the overall task as a set of somewhat smaller subtasks, which are each executed in full by a single PE. *Medium-grain* parallelization falls in between these two.

The optimal choice of granularity depends on the time required to transfer data between PEs and the time required to execute operations on a PE. If the ratio of communication time to execution time is high then a coarse-grain parallel algorithm should be used [Jose97].

On a network of Unix machines, memory accesses take on the order of nanoseconds whereas communication of a few bytes takes on the order of microseconds to milliseconds depending on traffic (assuming an Ethernet-based network). Thus, to improve the performance of the GA while minimizing communication overhead, we should choose a coarse-grain parallelization strategy.

2.4 The PRAM Model

The *parallel random-access machine* (PRAM) model provides a framework for theoretical development of parallel algorithms [ChTi96]. A PRAM is a MIMD *multiprocessor* system with an unbounded set of PEs, each with its own local memory, that communicate through shared memory. Shared memory accesses are assumed to take $O(1)$ time and the PEs are inherently synchronized. Although such a system does not exist, algorithms for the PRAM provide a basis against which other implementable algorithms can be compared.

Suppose that the running time of the fastest algorithm to solve a problem of size n, on a sequential machine, is bounded (below) by $\Omega(f(n))$. Given a parallel machine, with a constant number p of PEs, suppose the fastest algorithm to solve the same problem is bounded (above) by $O(g(n))$. For a PRAM, it can be shown that $g(n)$ cannot be asymptotically better than $f(n)$ [Jose97]. That is, $f(n) = O(g(n))$ always. Since most parallel architectures are no more powerful than a PRAM, this suggests that parallelizing an algorithm for a fixed number of processors can never asymptotically improve the running time over the running time of the best sequential algorithm for the same problem.

If we allowed p, the number of PEs, to grow with n, the input size, then asymptotic improvement becomes possible from parallelism. As such, the parallel algorithms that will be described here make the assumption that $p = O(n)$. Practically speaking, no architecture can support an indefinite growth in p. However, we could have p scale with n in a way that reflects the available number of processors and the likely range of n.

2.5 The Genetic Algorithm

The *genetic algorithm* (GA) is an algorithm to search a parameter space probabilistically for an optimal set of parameters. The algorithm is provided with a scalar *fitness* function $F(\mathbf{X})$ that operates on a vector of parameters $\mathbf{X} = [x_1, x_2, \ldots x_k]$. In a manner loosely based on biological evolution, the algorithm attempts to find the vector of parameters for which $F(\mathbf{X})$ is maximized.

The vector $\mathbf{X}$ is analogous to an individual in biology, its parameter x_i is analogous to a chromosome. The bits that make up x_i are analogous to genes. Table 2 provides a definition of the GA (there are others) [Jose97]. Extra parameters that must be provided are italicized. The population size, replacement ratio, and mutation rate are represented by *n*, *r*, and *m*. Operators to generate new individuals, test termination, perform cross-over and mutation are also needed. A *cross-over* of two individuals produces a new individual with characteristics of both parents. A *mutation* of an individual produces a new individual with a small gene difference.

Table 2. A prototypical genetic algorithm.

- *Generate n* individuals for a population **P**.
- Repeat steps (1) to (6) until a *termination condition* is met.

(1) Create an empty next generation $\mathbf{P}_{next}$.

(2) Evaluate the fitness of individuals in **P**. Individuals are selected in steps (3) and (4) according to probability $F(\mathbf{X})/\Sigma F(\mathbf{X})$.

(3) Select (1-*r*)*n* individuals from **P** and add them to $\mathbf{P}_{next}$.

(4) Select *rn* pairs of individuals and add the result of a *cross-over* operation to $\mathbf{P}_{next}$.

(5) Irrespective of fitness, *mutate* individuals in $\mathbf{P}_{next}$ according to probability *m*.

(6) Replace population **P** by $\mathbf{P}_{next}$.

- Return the individual with highest fitness.

In Table 1, we stated that a parallel GA should not violate GA fundamentals. We can identify three main principles of the GA. First, in each generation, the GA searches for an optimal individual along many different paths. Second, individuals contribute more to future generations if their current fitness is high relative to the population. Third, the genes and chromosomes of all individuals (the gene pool) are not isolated in pockets but recombine globally to produce new individuals.

2.6 GA for the PRAM

A GA for an *n* processor PRAM can allow each processor to own one individual in the populations **P** and $\mathbf{P}_{next}$. Because steps (1), (2), (5) and (6) of Table 2 do not involve interaction between individuals, each PE can operate on its own individuals in parallel to other PEs. These steps can thus be performed in *O(1)* time for one iteration of the loop, or one *epoch*.

By numbering the PEs from 1 to *n*, we can guarantee the correct proportion of replacement and cross-over by stipulating that the bottom (1-*r*)*n* processors perform step (3) and the top *rn* perform step (4). Except for the selection of one or two individuals by each PE, all of steps (3) and (4) can be executed in parallel. The selection of individuals is based on a fitness comparison between individuals and warrants further discussion.

An analogy of the probabilistic rule outlined in Table 2 is a roulette wheel of *n* sectors where the area of each sector equals the fitness of a distinct individual. The roulette wheel can be implemented as an array of sorted partial sums of all fitnesses [Jose97]. Selecting an individual involves picking a random number in $(0,\Sigma F]$ and searching the array for the position where the number falls between adjacent partial sums. Once this array is constructed, PEs can select one or two individuals in parallel since concurrent reads are allowed. With a binary search, this would take *O(log n)* time.

The array of partial sums can be constructed by the PEs in *O(log n)* time using the *parallel prefix* algorithm [CoLR89]. Therefore, one epoch of a parallel GA for the PRAM would run in *O(log n)* time. By comparison, a sequential GA must at least compute the fitness of *n* individuals, incurring a running time of $\Omega(n)$.

III System Design

3.1 Decentralization

The parallel GA for the PRAM is not implementable on a MIMD multicomputer environment. To design a GA for the Unix network, that meets the requirements of Table 1, decentralization was the first design choice.

A centralized algorithm would be one where certain computers on the network have a significantly different role than others. For example, a single computer may coordinate the activities of the other computers, distributing tasks and collecting results. The primary disadvantage of this approach is that if the coordinating computer fails, the whole system also fails. Electing a new coordinator often takes a considerable amount of time [SiGa95]. Another disadvantage is that the speed of the coordinator is a bottleneck for the system. Other computers may be capable of working faster than the pace set by the coordinator.

Consequently, a decentralized approach is selected for the network. A decentralized solution should be fault tolerant and adaptable to varying network loads.

3.2 Parallel Launch

The first step in executing any algorithm is to launch it. Given a pool of *p* machines, we would like to launch a process reliably on every single machine from any one machine. This *parallel launch problem* can be solved in either a centralized or decentralized manner. The centralized scheme has a *launch machine* running through a list of all processors and executing the required process remotely on each machine. This requires *$\Omega(p)$* time and if the launch machine crashes then potentially many machines will remain unlaunched.

A decentralized solution to the parallel launch problem is as follows. Let us call the program that is run on the launch machine, the *launcher*. The launcher simply launches itself on a random machine that it has not launched yet, and repeats this procedure until no such machines are left. Thus, when a machine is launched, it begins to launch other machines. To prevent multiple copies of the launcher running on every machine, which would be inconsiderate to other users, the launcher can abort if it is already running.

If a launcher process is shut down by a fault, all processors (that are not shut down) will still be launched providing that one launched machine remains. The launcher is therefore very fault tolerant. However, the launcher still runs in *$\Omega(p)$* time since it does not know that other processes are helping to complete the launch task. To remedy this, each launcher process can attempt to contact a remote launcher process before launching that machine. If it is successful, both processes exchange which machines they know have been launched. The launcher concludes that the launch task is complete when it knows that an attempt has been made to launch every machine, and not that every machine has indeed been launched (which helps to prevent deadlocks).

Consider that, at some point, the launcher has been executed on k of p machines, where k is small compared to p. Then chances are that each launched machine will attempt to launch a distinct unlaunched machine. Therefore, in the initial stages, the number of launched processes doubles every parallel round. As the number of launched machines grows, it will quickly become likely that launched machines will choose to launch other launched machines. However, the launched machines will also learn more quickly of the launch status of all machines and will narrow the pool of machines that they attempt to launch. Thus, we expect the running time of the algorithm to be *$O(\log p)$* on average, which is asymptotically faster than the centralized approach.

We shall call this algorithm the *parallel random launch algorithm* (PRLA). Note that, in principle, the PRLA is applicable to any decentralized parallel problem. Positive features of the PRLA are that it can handle MIMD multicomputer environments that are asynchronous, dynamic, and faulty, with good performance. However, these statements need to be justified by supporting experimental results.

3.3 Parallel GA Dilemma

Not all tasks can be implemented effectively in parallel. Some tasks are inherently sequential where the next subtask involves the results of the current subtask. We have demonstrated that it is theoretically possible to execute one epoch of the GA in *$O(\log n)$* time. However, the parallel GA for the PRAM uses a medium grain parallelization where each processor handles the execution of one individual in the population.

If we were to implement the PRAM solution on a Unix network, we would likely fail to improve the speed and search power of the GA. Networked computers typically do not share a global memory with *$O(1)$* access time and do not execute in a synchronous fashion. The time required to simulate the global memory and synchronization (using message passing, for example) will likely be large compared to the time spent evaluating the fitness of a single individual.

As discussed in Section 2.3, coarse grain parallelization should be employed on the Unix network. Having each processor manage a subset of individuals would better match the network characteristics. However, to recombine individuals for the next generation, each processor may have to communicate with every other processor since individuals are selected probabilistically from the whole population. The synchronization and communication required would be inefficient in a highly asynchronous, load varying network of computers.

The next milestone in granularity is one iteration of the GA on a whole population (or one epoch). For nontrivial problems (like *genetic programming* [Mitc96]), the time taken for one epoch will probably be sufficient to make cooperation over the network viable. By allowing an epoch to execute on a single computer, we eliminate the problems associated with probabilistic selection of individuals. However, at this level, the GA becomes a strict sequential problem since the results of one epoch are needed before the next epoch can begin.

Thus, we are faced with a serious dilemma. The solution lies in examining the GA carefully and redefining the algorithm so that coarse grain parallelism over the network becomes feasible.

3.4 Alternative GA Prototype

An alternative prototype for the GA, based more on sociology than biology, is readily brought to mind. Suppose that *colonies* of n_C individuals reside independently on each of the p processors in the network. The processors are analogous to *islands* in an ocean. Each colony performs the prototypical sequential GA of Table 2. After an epoch is completed on an island, let half the individuals in the colony (chosen randomly) board a *ship* and travel to another island where they are welcomed into the colony residing there.

This algorithm satisfies the three essentials of the

GA for a population of size $n = pn_C$ and allows for viable coarse grain parallelism. First, a population of n individuals is always examined. Second, at the colonial level, an individual is more likely to influence the next local generation if its fitness relative to the colony's total fitness is high. Because of the continual migration of individuals, the descendants of every successful individual will eventually end up in every colony and will compete against individuals from other colonies. Thus, an individual is more likely to influence future global generations if its fitness is high relative to the population's total fitness. The third essential of the GA is that recombination of genes occurs across the entire gene pool. Although only the genes of individuals within the colony are recombined in every colonial epoch, continual migration ensures the recombination of genes from diverse parts of the entire population.

Because the network of Unix computers is inherently asynchronous, each island will finish its colonial epoch at different times. However, if we integrate the characteristics of the network into the GA prototype then we will not require synchronization and its associated overhead. Let each island have a *harbour* which can hold one ship. When a ship of individuals arrives at a harbour, it remains there (without any evolution) until it is accepted by the island. After an island finishes its epoch and releases a ship into the ocean, it then accepts waiting individuals. If a ship at sea encounters a full harbour then it tries another island. The harbour is a buffer to handle the asynchronous nature of the network.

3.5 Trade Policies

The above discussion assumes that the genes from any colony will make their way to every other colony but this will depend on how ships select islands, which we shall call a *trade policy.* For example, policies could be defined where a group of islands form a clique in which individuals never find their way in or out.

An ordered trade policy that ensures genes from a colony make their way to every other colony is as follows. Number each machine from 0 to p-1 and have the ship leaving island k select island k+1 *mod* p. One problem with this policy is that it imposes specific (though symmetric) roles to each processor which may lead to problems concerning fault tolerance.

Let us define a *round* of the GA to be a period in which every processor completes at least one epoch. The duration of a round on the network will vary but, on average, it will take $O(n_C)$ time since epochs are executed in parallel. To best ensure the GA essentials are met, a trade policy should distribute genes from any one colony to every other colony in a minimal number of rounds. For example, the ordered trade policy described above requires $O(p)$ rounds to redistribute genes completely. If it were possible to redistribute genes in $O(1)$ rounds then we would be back to the GA of Table 2.

3.6 Optimal Trade Policy

We would like to know what the best possible trade policy is and the minimal number of rounds for gene redistribution. Consider a colony A amongst p total colonies. After one round, every colony will have sent individuals to another colony. Thus, the original gene pool of colony A will now be shared between two colonies A and B. After another round, the best possible scenario is for colonies A and B to send individuals to two colonies that are different from A, B, and each other.

Therefore, every round, we can at most double the number of colonies that contain genes (or descendants) from the original group of individuals in colony A. Since there are p colonies in total, it will take $O(log\ p)$ rounds for the original set of genes to be distributed completely. Note that this can occur simultaneously for all sets of genes in all original colonies. The optimal trade policy for four colonies is shown in Fig. 1.

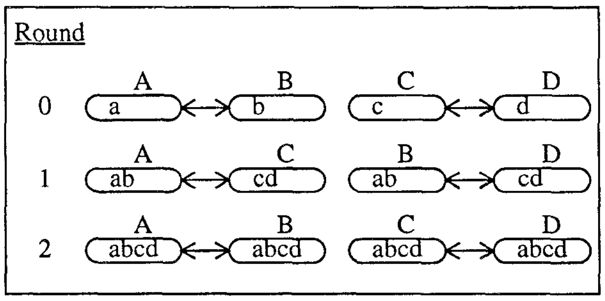

Fig. 1. Optimal trade policy for four colonies.

3.7 Decentralized Trade Policy

The optimal trade policy requires considerable coordination between colonies since each must carefully avoid certain other colonies every round. Implementing this coordination would either require a coordinator or an imposed ordering on the processes of some kind. Both of these approaches would cause problems for fault tolerant computing. Therefore, we must find an alternative decentralized policy.

A completely decentralized trade policy is for every ship at sea to select an island at random from those islands that it has not already visited. We shall attempt to show that this policy will take $O(log\ p)$ rounds on average to redistribute all genes in the population. To begin with, for a given colony A, we will determine how long it will take for the genes presently in colony A, or the *original genes*, to reach all p colonies.

Let k_i represent the number of colonies that contain original genes after i rounds with $k_0 = 1$. Given k_i, we can determine the expected value of k_{i+1}, or $E\{k_{i+1}\}$, as follows. After each round, every colony will have received a ship of individuals from some other colony. Because all selections are random, it is equally likely for a colony to have received a ship from any of the p colonies. Thus, in the i+1 round, the probability that a colony receives a ship with original genes is k_i/p. At the start of the i+1 round, there are p-k_i colonies which do not have original genes. The expected number of colonies in this group that receive original genes is approximately $(p-k_i)k_i/p$ which makes $E\{k_{i+1}\}$ equal to approximately $k_i+(p-k_i)k_i/p$. This equation is rewritten below.

$$E\{k_{i+1}\} \approx k_i + \frac{(p-k_i)\,k_i}{p} = 2k_i - \frac{k_i^2}{p} \qquad (3)$$

Equation 3 agrees with the intuitive idea that, when k is small relative to p, the number of colonies with original genes roughly doubles each round. This doubling tendency decreases as k increases since genes are more likely to be sent to colonies that have already received original genes. If we assume that the actual number of colonies after i+1 rounds that have received original genes is approximately equal to the expected number of colonies that have received original genes then we can obtain the recurrence relation of Eq. 4. Note that this assumption is not always reasonable because actual values may be quite different from expected values. However, we shall still make the assumption.

$$k_{i+1} \approx 2k_i - \frac{k_i^2}{p} \qquad (4)$$

Solving the above relation in terms of $k_0 = 1$ is difficult to do directly. However, after i rounds, we can define the number of *remaining colonies* (ones that have not received original genes) as $x_i = p$-k_i. Substituting this value into Eq. 4 leads to a recurrence relation (Eq. 5) that is more readily solved in terms of $x_0 = p$-1 [Jose97].

$$p - x_{i+1} \approx 2\,(p - x_i) + \frac{(p-x_i)^2}{p} \qquad (5)$$

$$x_{i+1} \approx \frac{x_i^2}{p} = p\left[\frac{x_0}{p}\right]^{2^{i+1}} \qquad (6)$$

Equation 6 is plotted in Fig. 2 for p = 100 colonies (and x_0 = 99). The graph demonstrates that almost all colonies have received original genes after 9 rounds. The curve never actually reaches zero because there is always a finite probability that a few colonies will take forever to receive the original genes from colony *A*.

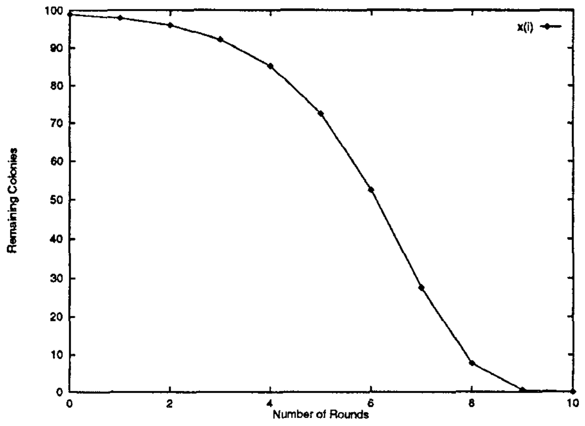

Fig. 2. Remaining colonies after i rounds.

The percentage of remaining colonies after i rounds is x_i/p. Using Eq. 6, we can find how many rounds are needed to reduce this percentage to some fixed percentage a [Jose97]. Equations 7 and 8 summarize such an analysis. Equation 8 predicts that if we had p = 100 colonies then it would take about i = 9 rounds for the percentage of remaining colonies to equal a = 1%. This result agrees with the graph shown in Fig. 2.

$$a = \frac{x_i}{p} \approx \left[\frac{x_0}{p}\right]^{2^i} = \left[\frac{p-1}{p}\right]^{2^i} \qquad (7)$$

$$i \approx \frac{\ln(-\ln a) - \ln[-\ln(1 - 1/p)]}{\ln 2} \qquad (8)$$

According to Eq. 8, for any fixed percentage a, the number of rounds that are required are bounded by *O(-ln(-ln(1-1/p))* which is equivalent mathematically to *O(log p)* [Jose97]. Although this result involved a few approximations, it is reasonable to expect that, on average, the number of rounds needed to distribute genes from an original colony *A* to most of the colonies will be bounded by *O(log p)*. Because this occurs simultaneously for all original colonies, the rounds required to redistribute genes completely will be similarly bounded.

Thus, a random trade policy yields a completely decentralized parallel GA with a performance similar to that of the optimal trade policy. We shall call this parallel GA the *parallel random genetic algorithm* (PRGA).

3.8 Trials at Sea

We stated in Section 3.5 that the time required for one round will be bounded, on average, by $O(n_C)$, which is the time required for one epoch in every colony (executed in parallel). However, this does not consider the time required to execute the random trade policy. When an island finishes an epoch, it sends a ship of individuals to another island. If that island's harbour is full then the ship will have to try another island. Our overhead increases if it takes many trials on average to find an island with an empty harbour.

Initially, all harbours are empty. After island *A* finishes an epoch, it sends a ship to island *B*'s harbour. The ship will have to wait in the harbour until island *B* has finished its epoch. In the interim, other ships that come to island *B* will have to try again. Island *C* then finishes an epoch and sends a ship to another island. However, island *C*'s harbour is empty. Its colony is now half its previous size and it cannot proceed unless it randomly generates new individuals. Let us require, however, that an island that finishes its epoch must wait for a ship to come to its harbour before starting the next epoch. We shall relax this requirement in Section 3.11.

Let *x* be the number of islands waiting for a ship to arrive in its harbour and let *y* be the number of ships waiting in a harbour to be accepted by the corresponding island. If *p* is the total number of islands then $x + y \leq p$ (since an island cannot be waiting for a ship with a ship waiting for acceptance in its harbour). There can also be islands with empty harbours that are not waiting for ships (they are simply in the middle of executing an epoch). When an island finishes its epoch, there are four possible scenarios that are outlined in Table 3.

Table 3. Possibilities for a finished island.

- The island sends a ship to an island waiting for a ship, and then waits for a ship to come to its own harbour (*x* and *y* do not change).
- The island sends a ship to an island waiting for a ship, and then accepts the ship waiting in its harbour (*x* and *y* are decremented).
- The island sends a ship to an island not waiting for a ship, and then waits for a ship to come to its harbour (*x* and *y* are incremented).
- The island sends a ship to an island not waiting for a ship, and then accepts the ship waiting in its harbour (*x* and *y* do not change).

In all scenarios, the number of ships waiting in a harbour and the number of islands waiting for a ship change in lock step. In other words, *x-y* is always a constant. When the PRGA is started, all harbours are empty and no island is waiting for a ship. Thus, *x-y* will initially and always equal zero. Put another way, the number of islands waiting for a ship will always equal the number of ships waiting in a harbour. Because $x + y \leq p$, then *x* and *y* are at most *p*/2 each.

What this implies is that, at any time, no more than half the total number of islands will have full harbours. Thus, the probability that a ship at sea selects an island that has a full harbour is no more than 0.5 since each ship selects an island randomly from all islands. Therefore, the expected number of trials a ship must make before finding an empty harbour is less than two. Consequently, the average running time of one round is indeed $O(n_C)$ since a round involves a parallel execution of an epoch on every island, which is $O(n_C)$, and a parallel trade of ships, which is $O(1)$.

3.9 Comparison to the PRAM Solution

The PRGA preserves the essentials of the prototypical GA of Table 2 but is considerably different from it. Thus, it is difficult to compare the PRGA with the prototypical GA for the PRAM. Using *n* processors, the PRAM GA operates on a population of size *n* and takes $O(\log n)$ time to execute one epoch. Using *p* processors, the PRGA operates on a population of size *n*, which is subdivided into *p* colonies of size n_C where $n = pn_C$. To properly compare the PRGA to the PRAM GA, *p* must grow linearly with the population size. A simple way to do that is to make the colony size n_C a constant (for example, 100 to 1000 individuals). Thus, $p = n/n_C$.

It seems reasonable to define a *complete epoch* of the PRGA, comparable to an epoch on the PRAM, as the time it takes for complete redistribution of genes. We have shown that this will take, on average, $O(\log p)$ rounds and have shown that one round takes, on average, $O(n_C)$ time. Thus, on average, one epoch of the PRGA should take $O(n_C \log p)$ time. Since $p = n/n_C$ and since n_C is a constant, the average asymptotic performance of the PRGA is $O(\log n)$ which is comparable to the exact asymptotic performance of the GA for the PRAM.

3.10 Restarting Shut-down Computers

The PRGA uses the PRLA as a launch vehicle. The PRLA quickly launches all computers in the pool that are not shut-down or that do not shut down during the launch process. Once the launcher, running on each launched computer, concludes that the launch is complete, it executes the PRGA. Suppose that once the PRGA is begun, computer *A* in the pool shuts down. After some time, computer *B* executing the PRGA will attempt to communicate with computer *A* (to deliver a ship of individuals). Upon failure, computer *B* will know that computer *A* either has terminated or has not launched yet. If we modify the PRGA and PRLA slightly, we can show that computer *B* simply has to execute the launcher on computer *A* to restart it.

We modify these algorithms so that the PRGA can receive and identify a connection from the PRLA and respond with a special message, *beginGA*. When the launcher process running on any computer receives the *beginGA* message, it immediately concludes that the PRLA is complete and begins to execute the PRGA. The only change this will make to the initial launch sequence is that it will become faster. In the initial launch sequence, the first launcher process to enter the PRGA will have correctly concluded that a launch attempt has been made on every computer. Another launcher process that connects to this process will now receive a very quick reply to let it know that a launch attempt has been made on every computer. This modification also means that a restarted computer will not go through the entire PRLA. Once it connects to a computer running the PRGA, it will immediately begin the PRGA.

If a computer shuts down due to a system crash or for maintenance, then it likely will not be available for a significant amount of time. Therefore, each connection request that comes in this period from the PRGA will result in a launch attempt. To reduce this communication overhead we can modify the PRGA further. Let each computer in the PRGA maintain a list of all computers in the pool along with some status information. Initially, the state of each computer is marked as *ALIVE*. Whenever computer *A* successfully connects to computer *B*, both computers mark each other as *ALIVE* in their status lists. If computer *A* is unsuccessful in making a connection with a computer *B* that is marked *ALIVE*, then computer *A* runs the launcher on computer *B* and marks the state of computer *B* as *RESTART* = 1, *WAIT* = 1.

Computer *A* uses the counter *RESTART* to count how many times it has attempted to restart computer *B*. The counter *WAIT* indicates how many colonial epochs computer *A* is to completely ignore computer *B*. Thus, after each epoch, computer *A* will decrement the value of all non-zero *WAIT* counters in its status list. In addition, when computer *A* is selecting another computer (to send a ship of individuals), it will only select from those computers marked *ALIVE* or that have *WAIT* = 0. If computer *A* is unsuccessful in connecting to computer *B* that has *WAIT* = 0, then it runs the launcher on computer *B* and increments the *RESTART* counter for computer *B*. To ensure that it waits longer this time, it sets the counter *WAIT* to equal the value of the counter *RESTART*.

This protocol allows the PRGA to ignore computers that cannot be restarted, but to still attempt to restart them periodically (the period simply grows). However, as discussed in the next section, there is still another modification to the PRGA that is necessary to ensure that it will continue to function in *O(log n)* time.

3.11 Deadlock, Starvation, and Congestion

The random nature of the PRGA itself makes it suitable for fault tolerance. For example, if a colony suddenly disappeared from the population, then the PRGA can still function meaningfully. Although the tasks performed by the colony was useful to the overall task, it was not crucial. Similarly, if a ship of individuals never reaches an island, the overall task should still be able to proceed. Not all algorithms have this graceful degradation property where all subtasks do not need to be completed so long as most are completed.

The PRGA, as we have designed it so far, cannot proceed effectively if some colonies disappear or new ones appear. This is because a possibility of deadlock, starvation, and congestion exists if the network is faulty. We required that every island which finishes an epoch must wait until a ship arrives in its harbour before continuing with the next epoch. In a faulty system, all islands that are in the middle of an epoch may crash. This would leave the waiting islands in a deadlock. Starvation arises when all islands with an empty harbour crash. In that case, the next island to finish an epoch will not be able to send its ship to any harbour.

The random trade policy executed in *O(1)* time because the number of islands waiting for a ship and the number of ships waiting in a harbour changed in a lockstep fashion. However, in a faulty system, this lock step may be broken, resulting in ships trying many harbours before finding an empty one. This congestion would dramatically decrease the performance of the PRGA.

To eliminate two of these problems, we use the property that, on average, it should take no more than two tries to find an empty harbour. Each island can tabulate the average number of trials its ships took to find an empty harbour in recent epochs (for example, over the last ten epochs). When an island finishes an epoch, it finds an empty harbour for its ship and updates the average number of trials. However, it will send its ship to that harbour only if the average is no more than two. If the ship is not sent and if there is no ship waiting in the island's harbour, then the individuals are returned to the colony for the next epoch. If the ship is not sent and a ship is waiting in the harbour, then the outgoing ship is sunk and the individuals in the harbour are welcomed into the colony. This mechanism will detect starvation (no empty harbour) and congestion (few empty harbours) and will compensate (reduce full harbours).

To eliminate the possibility of deadlock, we will not require that an island wait for a ship to enter its harbour before continuing with the next epoch. If its harbour is empty then it will randomly generate individuals to fill the vacated half of the colony and continue on. Without the mechanism of the previous paragraph that removes individuals, the mechanism described here would gradually increase the proportion of full harbours (since it increases the total individuals in the system without changing the capacity of the colonies). The mechanism described here also guarantees that each computer in the PRGA will continue immediately with the next epoch after finishing the current epoch. It will also introduce new genetic material into the system.

3.12 Parallel Termination

The PRGA must have a termination condition. The condition may be to stop either after a certain amount of time has elapsed or after a certain number of epochs have been executed by a colony. It may also be when one of the colonies produces an individual with a fitness above a certain threshold. When the termination condition is met somewhere in the network, we would like all processors to terminate quickly saving their best individuals to a file. If the network supports a distributed file system then each processor can append their best individual to the same file and the user will only have to look at the results stored in one file. However, we will not consider how the results are collated and will only consider the termination process.

Termination can be accomplished in a similar manner to launching. A processor, that knows the termination condition is met, randomly selects another processor that it knows has not received the termination message, connects to that processor, and informs it to terminate. If it connects to a machine that has already received the termination message then they exchange the list of processors that they know have received the message. A processor terminates when it knows that every processor either has received the termination message or could not be contacted (it has shut down). It is expected that this scheme would distribute the termination message in $O(\log p)$ time where p is the number of processors.

A pitfall in this approach is due to the fault tolerance of the PRGA. Suppose processor A attempts to inform processor B of termination but cannot make a connection. It will then assume that processor B is down and therefore proceeds with the next machine. However, another processor may subsequently restart processor B before receiving the termination message itself. Processor B may subsequently restart other computers that have terminated (processor A may have terminated by now). Another problem arises when all machines that currently have the termination message crash. In this case, the termination message will have been lost and the processors that crashed may be restarted later.

The solution to this problem is for each processor to use local non-volatile memory to store the termination message once received and then proceed to inform other processors. If it is restarted, it can examine the non-volatile memory and realize that some machine is still executing despite termination. It can then repeat the termination algorithm. However, as a fail-safe mechanism, a time limit should be imposed when the system is initially launched, after which all processes terminate.

3.13 Implementation Remarks

A significant portion of the design has been implemented using the C++ programming language and the object-oriented paradigm [Jose97]. However, only a few remarks are made here about implementation.

C++ classes were constructed to abstract the elements of GA individuals and populations. The *GA individual* class was designed so that elements of the GA that change from problem to problem are contained within the class. However, the *GA population* class can be reused unchanged from one GA problem to the next.

For the PRLA and the PRGA, communication between machines was done using Berkeley sockets. These algorithms can be implemented using two concurrent processes on each machine, a server and a client. The server waits for a connection to which it responds in a concurrent fashion (ensuring fast response). The client process performs the active work of the PRLA and PRGA and connects to remote servers. Once the client finishes the PRLA, it proceeds to execute the PRGA. The server and client processes on each machine communicate using shared memory and semaphores.

Unix signals were used to help provide fault tolerance and termination over the concurrent processes on each machine. If either the server or client crashes on a machine then we want the other process to exit as well. By establishing a parent-child relationship between the server and client, signals can be automatically sent by the Unix kernel to help with this task.

IV NETWORK CONSIDERATIONS

Because the system was not fully implemented, it was not possible to test many things. However, a network load analysis was performed and is discussed here. This analysis helps confirm some of the underlying assumptions behind this proposal.

4.1 Network Load Analysis

Table 1 stipulates that a parallel GA for the Internet should be considerate to other users. A related question that we may ask is whether there is enough free CPU resources on the network for a parallel GA to be viable. We addressed this question by analysing the loads on the

University of Manitoba network. The analysis also provided justification for the requirement of fault tolerance.

Analysing the loads on a network is difficult to do from scratch. However, the university's computer services collects information on the average number of jobs running on each machine in the primary network and updates a list of this information every twenty minutes. This list is available to any user. To analyse the network loads for a large period of time, we wrote a script that would repeatedly sleep for twenty minutes, read the list of network information and append it to a file.

The script produced several megabytes of information within a week. In its raw form, however, this information was not readily useful. Because it was impractical to examine the information by hand or to analyse it with a spreadsheet, we wrote programs to analyse and summarize the data [Jose97].

4.2 Results of Load Analysis

The load average amongst machines from January 16 to January 22, 1997, is shown in Fig. 3. The ticks along the date line correspond to midnight of that day. Note that although the network load average is irregular, there are still patterns within it.

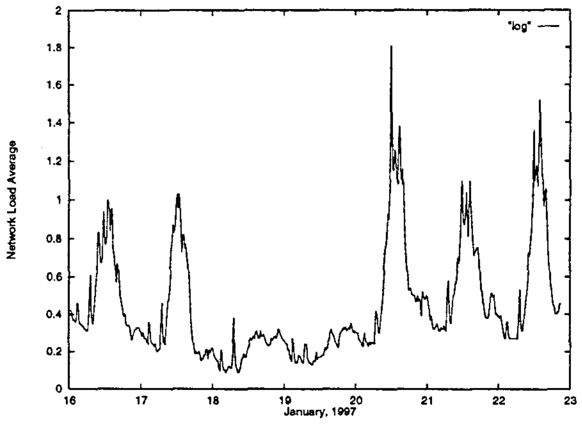

Fig. 3. Network load average for one week.

One pattern is that usage of the network peaks around midday (when many people check e-mail on their lunch break). As the day passes, the network usage decreases and remains low until well into the morning. The weekend also shows a marked drop in network usage (January 18 was a Saturday). From this data alone, one might conclude that during weekday afternoons there isn't enough network CPU time available (the average number of jobs in the run queue of a processor indicates how much the CPU is being shared amongst competing processes). However, Fig. 3 does not show the considerable variation in load averages between computers. Figure 4 shows the standard deviation of the load averages for the same period.

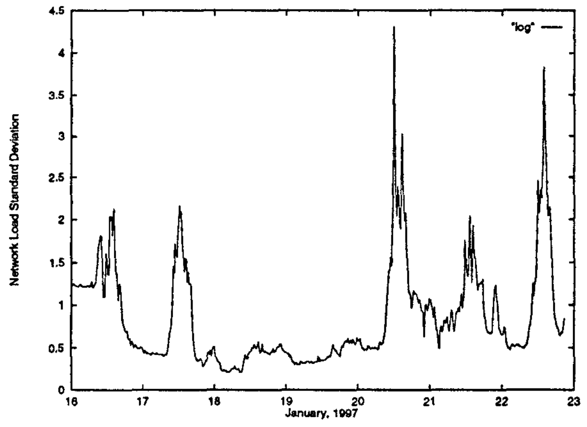

Fig. 4. Network load standard deviation for one week.

Figure 4 shows that there is considerable variation in load averages especially when the network load average is high. While some computers are being heavily used, there are many others that are relatively free. If the PRGA were to be implemented on this network then, during the times of slow activity in one area, the other computers will still keep going at a reasonable rate. Because the PRGA does not depend on a particular machine finishing its task before proceeding, it can handle the network dynamics. To be more considerate, we can run the algorithm with a lower priority. Thus, CPU usage of the PRGA will vary as the user load changes.

The reason that figures 3 and 4 end on January 23 was not because the script was stopped on purpose, but because the machine that the script was running on was shut down for maintenance. Logs were collected for subsequent weeks on different machines but on average the process was abruptly stopped within a week. More proof that machines are frequently taken out of the available pool is provided in Fig. 5.

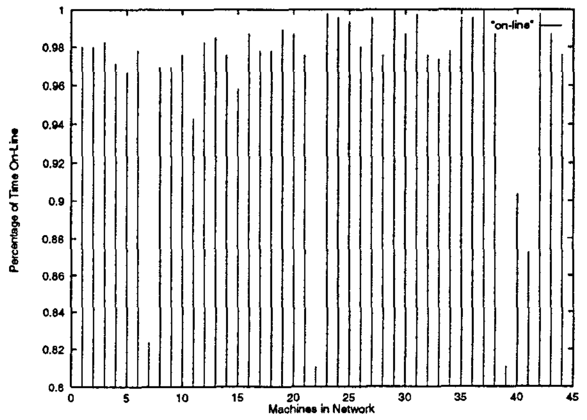

Fig. 5. Machine on-line percentage for one week.

Figure 5 shows the percentage of time each machine in the network was on-line during the week of January 24 to January 30, 1997. This data was obtained by counting the number of times each machine appeared in the log file for that week and normalizing the results. If there were no faults in the network then all machines would be on-line 100% of the time. The figure suggests that, within this week, there were a considerable number of shut-downs. However, because all on-line percentages are between 80% and 100%, we can conclude that machines that were shut down were soon restarted. Thus, it was wise to design a system that continues despite partial network shut-down and that attempts to restart shut-down machines. If this was not done, then the PRGA would not run effectively for one week on the University of Manitoba Unix network.

V Conclusions

We have discussed how a parallel implementation of the *genetic algorithm* (GA) on the Internet could improve the performance of the algorithm and have proposed a design for such an algorithm. This research is motivated by the use of the GA to solve many complex problems in realistic time frames where a faster GA could increase the pace of such research.

To begin with, we identified several requirements of a parallel GA for the Internet. The design presented here addressed most of these requirements.

Using the *parallel random access memory* model (PRAM) as a theoretical basis, we suggested that the best parallel implementation of an algorithm cannot be asymptotically better than the best sequential implementation of the algorithm unless the number of processors grows with the problem size. Using this assumption, we developed a parallel GA for the PRAM that is identical in function to the sequential prototypical GA. The running time for one epoch of this algorithm is $O(\log n)$, where n is the population size, an improvement over the $\Omega(n)$ running time for the sequential prototypical GA.

While the parallel GA for the PRAM provided a useful theoretical basis, we did not restrict ourselves to developing algorithms for ideal architectures. Instead, we applied parallel processing theory to the problem of implementing the GA on the Internet.

The concept of granularity was used to delimit possible solutions. Coarse-grain parallelization is the only practical way to avoid overhead costs in a MIMD multicomputer environment, as is the Internet. However, coarse-grain parallelization is infeasible with the prototypical sequential GA. To overcome this dilemma, we identified the essentials of the GA and proposed an alternative model, which does not violate these essentials, that is suitable for coarse-grain parallelization.

The alternative model, called the *parallel random genetic algorithm* (PRGA), divided the population into colonies residing on each machine. After executing an epoch, half the members of a colony migrate to another colony using a random trade policy and buffering. This procedure occurs in parallel over all colonies.

Through asymptotic analysis and some assumptions, we showed that the PRGA takes, on average, $O(\log n)$ time to execute an epoch. This performance is comparable to the exact asymptotic running time of the PRAM GA and is an asymptotic improvement over the $\Omega(n)$ running time of the sequential prototypical GA.

We developed a *parallel random launch algorithm* (PRLA) which matches the characteristics of the Internet and can be used to launch decentralized asynchronous algorithms like the PRGA. The running time of the PRLA was estimated to be $O(\log p)$, where p is the number of processors that need to be launched, an improvement over a centralized $\Omega(p)$ approach.

A considerable portion of the design presented here has been implemented and was briefly described. Experimental results were provided to justify the availability of CPU resources on the network and the need for fault tolerance. Future work could include completing the implementation and running experiments to confirm or refute the predicted performance.

References

[ChTi96] A. Chalmers and J. Tidmus, *Practical Parallel Processing*. London: International Thomson Computer Press, 1996.

[CoLR89] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill Book Company, 1989.

[Jose97] D. Joseph, "Design and Implementation of a Parallel Genetic Algorithm on the Internet," *B.Sc. Thesis*. University of Manitoba, March, 1997.

[Kins96] W. Kinsner, "Digital Systems Organization," *Lecture Notes*. University of Manitoba, 1996.

[Mitc96] T. M. Mitchell, *Machine Learning* (draft copy). New York: McGraw-Hill Book Company, 1996.

[Posk97] H. Poskar, "Data Dependence and Cache Coherence," *Lecture Notes*. http://www.ee.umanitoba.ca/~hpos/notes.html, February, 1997.

[SiGa95] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Massachusetts: Addison-Wesley Publishing Company.