Genetic Adventures in Parallel: Towards a Good Island Model under PVM

Keith Vertanen
Department of Computer Science, Oregon State University
303 Dearborn Hall, Corvallis OR 97330 USA
vertanen@cs.orst.edu

Abstract

Genetic algorithms (GAs) have proved to be a very useful and flexible way to solve difficult combinatoric problems. Arriving at high quality solution however often involves a very large number of evaluations and consequentially is quite computationally demanding. Evaluating GAs in parallel is thus desirable, but specialty parallel computers are not available to many who might benefit from parallel GAs. In this paper we will seek a parallel GA implementation under the PVM (parallel virtual machine) environment. After looking at the various parallel models available, we will choose the island model as the most appropriate for use with PVM. From a simplistic and inefficient starting implementation, we will develop a better implementation that provides close to optimal speedup.

1. Introduction

Genetic algorithms can be effectively employed to solve a variety of difficult problems. An example of a problem that GAs are employed to solve is the job-shop scheduling problem [8]. In the JSSP a number of jobs must be processed by a number of machines subject to various constraints and costs. Most practical scheduling problems can be reduced to the JSSP.

Another popular problem solved by GAs is time tabling [3]. The time table problem involves scheduling various resources subject to a number of hard constaints that cannot be violated. In addition, we often want to optimize a number of soft constraints that improve the "quality" of our schedule.

These sorts of problems occur frequently in the real-world, one might want to schedule maintenance activities during a power plant outage or schedule classes at a university. Often the organizations wishing to solve these problems don't have the specialized parallel computing resources that many of the parallel GAs require. Most organizations however do have a network of workstations available, a perfect platform for using the PVM parallel environment.

Under the assumption that we'd like to use PVM, we will look at the various models available for doing GAs in parallel. Once we've established the island model as the most reasonable option, we will develop an efficient implementation using a freely available C++ GA library.

In section 2 we will give a basic overview of genetic algorithms. Section 3 provides a brief introduction to the PVM system. In section 4 we will describe the different forms of parallelism available to GAs. Section 5 will describe the original island model software and analyze its performance to discover its weaknesses. In section 6 we will propose a new asynchronous software solution and describe the various design decisions made. Section 7 gives the results attained running the new software solution.

2. Genetic Algorithms

The idea of genetic algorithms is based on the evolutionary principles developed by Charles Darwin [4]. The first person to adapt this concept to artificial systems was Holland [7]. We represent solutions to a problem by a collection of individuals in a population (an individual might be a vector of values or a string of bits). We define a function that determines the fitness of an individual. Using this fitness value, we can choose individuals for reproduction. When individuals reproduce, their information is combined in some manner by a crossover operator. Individuals can also undergo a mutation, a random change in their genetic information.

The traditional genetic algorithm loop is as follows: (taken from [9])

1. Randomly initialize a population of individuals. (the first generation)
2. Evaluate each individual in the population.
3. Select parents of individuals according to some selection schemes.
4. Create new individuals by mating current individuals; apply crossover and mutation operators.
5. Delete chosen member of the population to make room for the new individuals.
6. Evaluate the new individuals and insert them into the population.
7. If time is up, stop and return the best individuals; if not, goto 3.

3. PVM

PVM, parallel virtual machine, is a software system to allow a network of heterogeneous workstations to be pooled together to work on a common problem [6]. PVM is a collaborative venture between Oak Ridge National Lab, University of Tennessee, Emory University and Carnegie Mellon University. Since its release in 1991, PVM has become one of the leading packages used for parallel distributed computing.

PVM provides a set of message passing functions, allowing tasks on different machines in the PVM network to communicate. PVM supports a non-blocking send and both blocking and non-blocking receives. The PVM communication functions insulate the programmer from the details of data format conversion, network routing, and other network specific details. With PVM, a programmer can develop one parallel program and port it to virtually any unix based architecture.

4. Parallelism in Genetic Algorithms

One of the frequently cited advantages of using GAs is their “natural” parallelism. Unfortunately the canonical version, with a global population undergoing global selection and crossover, does not parallelize well. The problem stems from the traditional selection algorithms such as fitness proportional, rank proportional, or truncation selection [5]. All these methods require global calculations to take place thus incurring high communication costs in a parallel implementation. It is possible to implement a global GA in parallel, but often requires specialized parallel computers [1] and certainly would be difficult to do under PVM.

Several approaches grew out of the desire to easily parallelize GAs, one is the cellular model. The cellular model [11] seeks to exploit fine grained parallel architectures. One individual is assigned to each processor. Selection and crossover is restricted to local neighborhoods of a particular processor. One again this model requires specialized massively parallel computers, not a very appropriate model for use under PVM.

The second approach is the parallel island model. The island model is designed to exploit a coarse grained architecture. Each processor is given a population of individuals. The processors evolve their populations using a serial GA. Periodically a processor may migrate a number of its individuals to another population. The amount of communication involved in the island model appears to be much more manageable than the global or cellular models. We will choose this as our candidate for a PVM implementation.

5. Original PVM island model software

To provide a basis for our GA development, we choose the GAlib[10] software package. GAlib is a library of C++ GA components. It provides not only simple GAs, but also an island model GA. An example program is included that implements the parallel island model using PVM message passing routines. It is this basic example that we use as a starting point for our exploration.

The workings of the example program can be summarized in the following steps:

1) Master spawns a slave process for each island population
2) Slaves create population of individuals
3) Slaves evolve their populations for a number of generations dictated by master.
4) Master waits for all slaves to signal they have finished.
5) Master chooses a random slave to migrate a random number of its best individuals to another random process.

6) Master collects populations from all slaves and compiles statistics.
7) Repeat from step 3.

The example genetic algorithm evolves a one-dimensional binary string of alternating zeros and ones. This is a toy problem, but it will allow us to focus on the operation and efficiency of the parallel island model under PVM rather than the details of a complicated problem.

Our first data was obtained with the following parameters:

binary string length	2048
individuals in each population	30
number of epochs*	50
number of generations per epoch	10
mutation probability	0.01
crossover probability	1.00

* An epoch is how many generations we allow the island process to evolve between updates to the master (10 in this case).

The PVM virtual machine consisted of 45 HP9000 workstations. The fitness function gives one point for every 0 bit in an even location in the string, and one point for every 1 bit in an odd location. The maximum score in our test runs would be 2048. The populations on the slave processes used a steady state GA.

In the first experiment, the number of populations was varied from 1 to 10. The runtime and the maximum fitness value achieved were the results tracked.

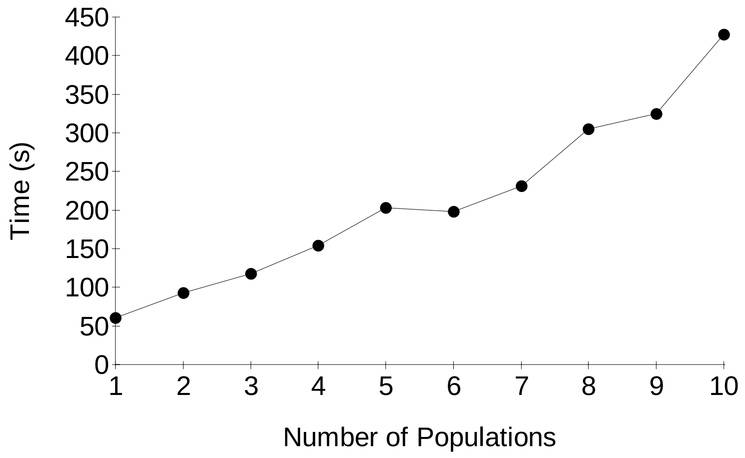


Figure 1
Runtime of the original software as the number of populations increases. Note that the total number of generations is also increasing proportional to the number of populations.

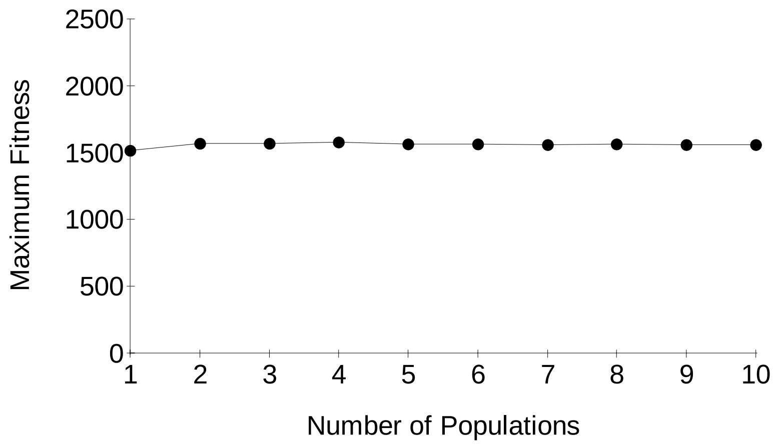


Figure 2
The maximum fitness attained as the number of populations was varied.

Both of these graphs are downright alarming. The first says that the more populations/machines we add, the slower things get. Note that as we add populations, we are not reducing the number of individuals or epochs. For one population, we do 50*10=500 total generations. For five populations, we do 50*10*5=2500 generations. But with five populations, we have five machines working on the generations. Ideally it should take the same time regardless of the number of populations. Our data is far from this ideal, a population of ten requires seven times the processing time of a population of one. We are gaining only a little over doing all the computations on a single machine. We will see why it performs this badly shortly, but first let us examine the second figure.

The second graph is even more disheartening, even with more time and machines invested in the problem, we get around the same maximum fitness value. If instead of looking at the maximum value, we examine the average fitness of the population, we obtain better results as shown in figure 3.

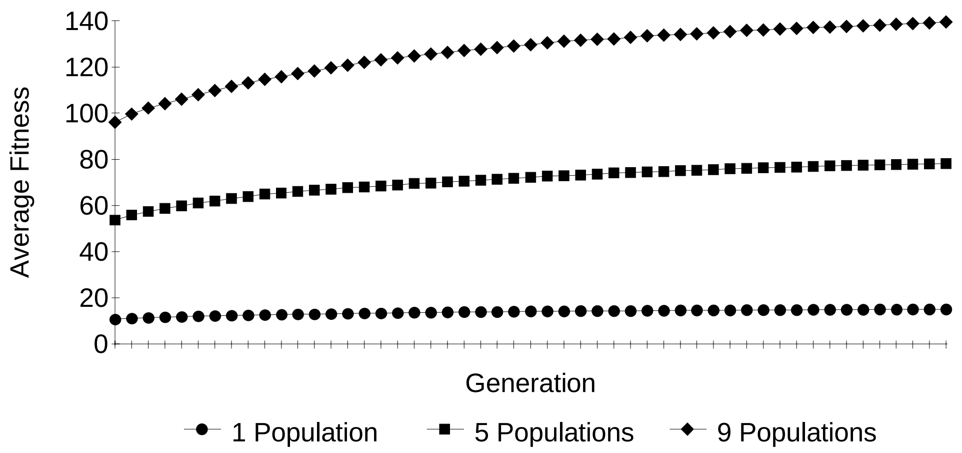


Figure 3
Average fitness as time progresses for three different numbers of populations.

Figure 3 shows how the average fitness of the population improves as we add more island populations. This perhaps in some sense “better”, at least it shows improvement as the number of populations increase. Our goal in this paper is not a thorough analysis of the fitness performance of the island model. The island’s model ability to outperform the serial GA has been established in other papers [2] [9].

Our main concern is why the performance degrades so noticeably as we add machines and populations (figure 1). The slave’s code included a blocking receive in which it waited for a command from the master. Essentially all slave tasks were being synchronized after each epoch. After all slaves had completed their epoch, one slave was chosen to exchange migrants with another. All other slaves had to wait for this migration to take place. After the migration, all slaves sent their populations to the master to allow global statistics to be generated. There seems to be quite a lot of waiting going on around here. The slave code was instrumented to time the amount of time spent doing useful work and the time spent on a blocking receive.

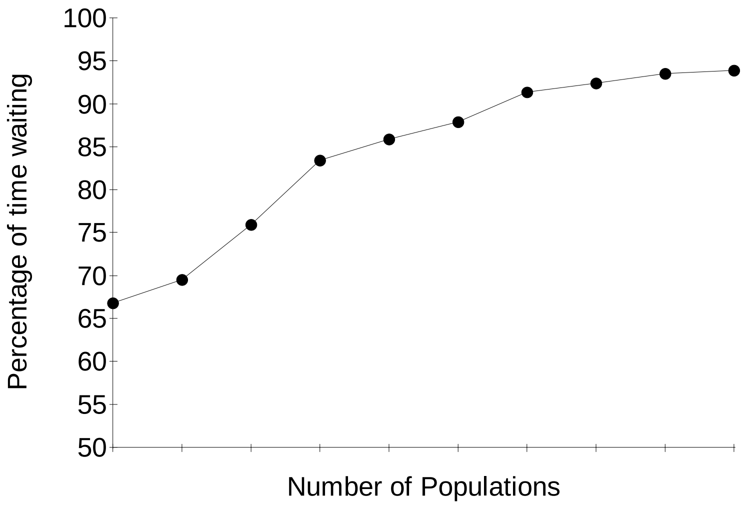


Figure 4
Average percentage of time each slave was doing a blocking receive.

Figure 4 shows that as the number of populations/machines increase, so does the percentage of wasted time. By ten populations, the slaves are spending 95% of their time waiting for messages. This is not surprising considering that the software is waiting to synchronize all slaves several times per epoch.

6. New and improved PVM island model software

The following goals were outlined for an improved software solution:

1) Avoid using blocking receives
2) Minimize communication costs

3) Allow global termination condition
4) Generate global statistics about population

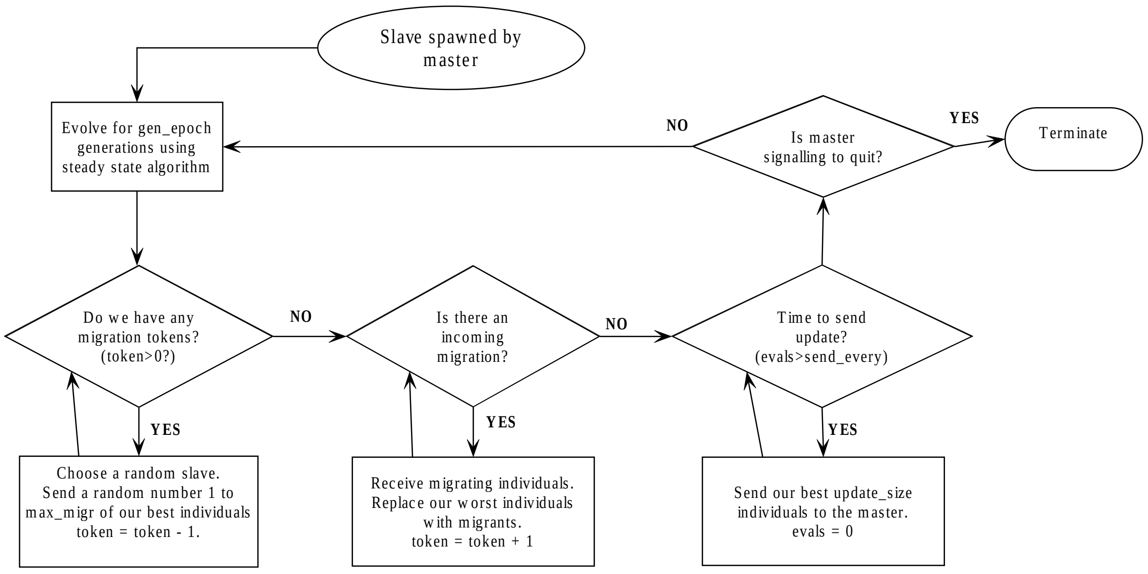
Goal 1 is achieved by having the slaves use only the non-blocking version of the PVM receive function. If a message is waiting, it will be received and appropriate action taken (namely either receiving a migration from another slave, sending a population update to the master, or terminating). The basic methodology here is this: if nobody wants anything from you, work on evolving your own population.

To achieve goal 2, we must find a communication efficient method for doing migration and for updating the master's population. In the original software, the master choose the slaves who were to be involved in migration. We would like to leave the master out of the decision. Towards this end, we will introduce the notion of a *migration token*.

When a slave posses a token, it must send out a migration of its best individuals after its current round of evolution. If it has multiple tokens, it will send out multiple migrations to other slaves processes. The master will send out a user specified number of migration tokens randomly to the slaves when they are spawned. After the first round of evolution, each slave posses as many tokens as the number of migrations it received during the previous round.

In order for the master to determine when to stop the evolution, it must periodically receive updates from the slaves. We will have the user specify the number of epochs each slave should perform between sending an update message to the master. The master can monitor the total number of generations evaluated so far and send the signal to quit when the desired limit is reached. Note that setting this value too high can cause our slaves to do too much work. For example, if we want 20,000 total generations and we tell 40 slaves to report every 5,000 generations, we only need the first four responses, the rest are wasted. Conversely, setting this value too low can cause an excessive amount of message traffic between the slaves and the master.

Achieving goals 3 and 4 while not violating goal 2 proved to be very difficult. For our sample problem of finding a bit string of alternating 1's and 0's, every individual sent over the network resulted in a 2K message. If we get carried away sending large populations of individuals between processes, performance can degrade alarmingly as the PVM message buffers become swamped with messages. Eventually one can even break PVM as the buffers exceed available memory.


Slave spawned by master
Evolve for gen_epoch generations using steady state algorithm
NO
Is master signalling to quit?
YES
Terminate
Do we have any migration tokens? (token>0?)
NO
Is there an incoming migration?
NO
Time to send update? (evals>send_every)
YES
YES
YES
Choose a random slave. Send a random number 1 to max_migr of our best individuals token = token - 1.
Receive migrating individuals. Replace our worst individuals with migrants. token = token + 1
Send our best update_size individuals to the master. evals = 0

Variables:

gen_epoch	Generation per epoch
max_migr	Max number of migrants
evals	Number of evaluations since last update sent
send_every	How many evaluations between updates
token	Number of migration tokens held
update_size	Number of individuals sent to update master

Figure 5
Flowchart of the operation of our improved slave task.

We opted to allow the user to decide how to balance their needs between good global statistics and termination and low message overhead. Besides allowing the user to specify how often the slaves should send population updates, the user can also specify how many of a slaves best individuals are sent to the master. If the master is set to terminate upon reaching a certain maximum fitness value or after a certain number of evaluations, sending just the best individual from each slave is sufficient. If however we wish to monitor the average fitness of all individuals in all populations, the slaves would need to send a complete population set to the master.

In order to control the size of messages between slaves, we also introduce a user specified maximum number of individuals to migrate (the original software choose a random number between 1 and the size of the population). We will choose a random number of the best individuals between 1 and this maximum value.

The only point of synchronization is upon completion of the desired number of generations. The master signals all slaves that it is time to quit. The slaves must respond before the master quits. This is perhaps not strictly necessary, the master could kill the slaves without signaling. We thought it was worth incurring this small communication penalty in order to insure a clean shutdown.

7. Results using New Software

Did we succeed in reducing waiting time and communication overhead? The new software was run with settings attempting to approximate the behavior of the original software. The master was told to terminate upon receiving N*10*50 generations, where N is the number of populations (50 was the number of generations each population did in the original run and 10 is the number of evaluations per epoch). One migration token was introduced into the populations since in the original run only one slave would be chosen to migrate after each epoch. The master requested population updates from the slaves every 200 evaluations. The slaves sent their entire population (30 individuals) during each update. The maximum number of migrants was set at 30.

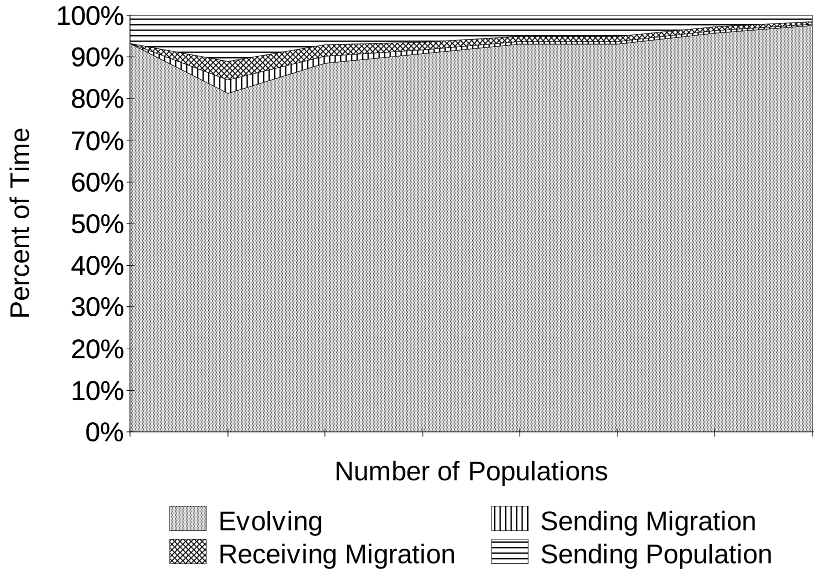


Figure 6
Average percentage of total time each slave was spending on various tasks.

It appears we have succeeded in minimizing communication overhead for at least a small number of populations (up to 10). The next step taken was to scale up to larger numbers of populations (up to 45). Unfortunately, using the above settings causes us to incur a catastrophic message traffic jam for large populations (as mentioned in section 6).

To prevent this message overload, we choose the following settings:

Migration tokens	1
Maximum migrants	5
Number of individuals sent to master	1

Other settings include:

Number of generations sought	200000
Number of epochs between sending updates	1000
Number of populations	1 to 45

All other parameters are as in the first two examples.

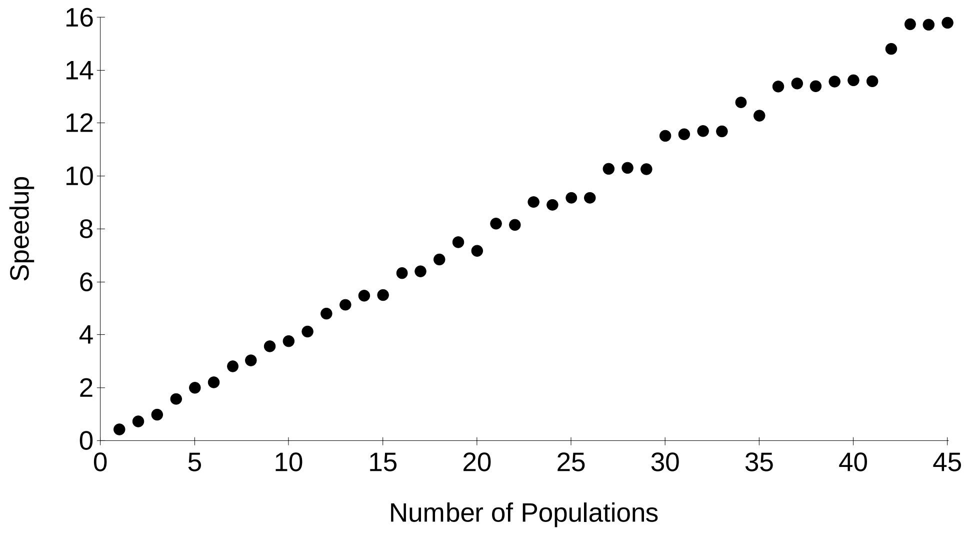


Figure 7
Parallel speedup of our improved algorithm when compared against a single population steady state GA.

In figure 7 we compare the runtime of our parallel software against the runtime of a serial steady state GA. Speedup is defined as the serial GA time divided by the parallel software time. The serial GA created one population of 30 individuals and ran for 200,000 generations. The optimal speedup would be equal to the number of populations. We achieved a linear speedup curve, but our speedups are one-third of the desired optimal values. Our comparison with a serial steady state GA is not entirely fair. Our software is a parallel implementation of the island model, a comparison against a serial island model is perhaps more appropriate.

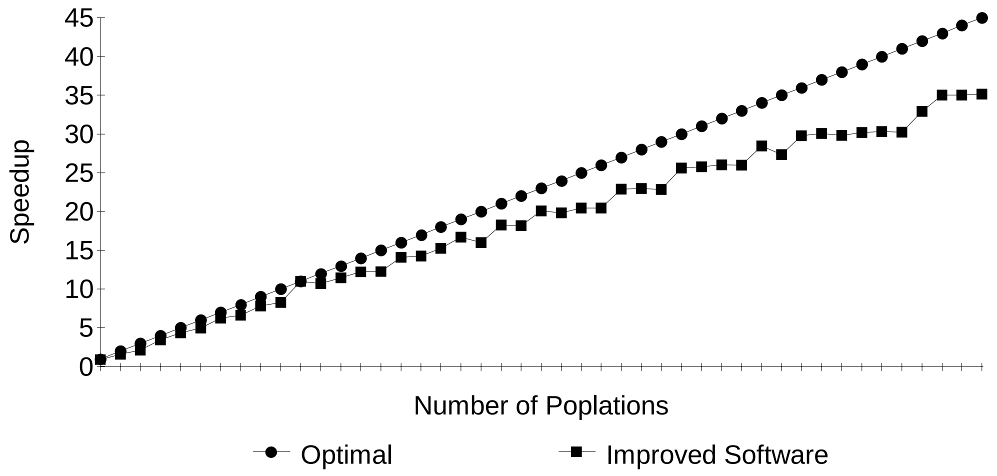


Figure 8
Parallel speedup of our improved algorithm when compared against a serial island model GA.

In figure 8 we compare the runtime of our software against the runtime of a serial island model. We used the GADemeGA class provide by GAlib. This class evolves multiple independent populations with a set amount of migration between all populations. As the class performed migration every generation whereas our software migrated every 10 generations, we turned off the migration for a conservative time estimate. Our software again shows linear speedup, this time within one-eighth of the optimal.

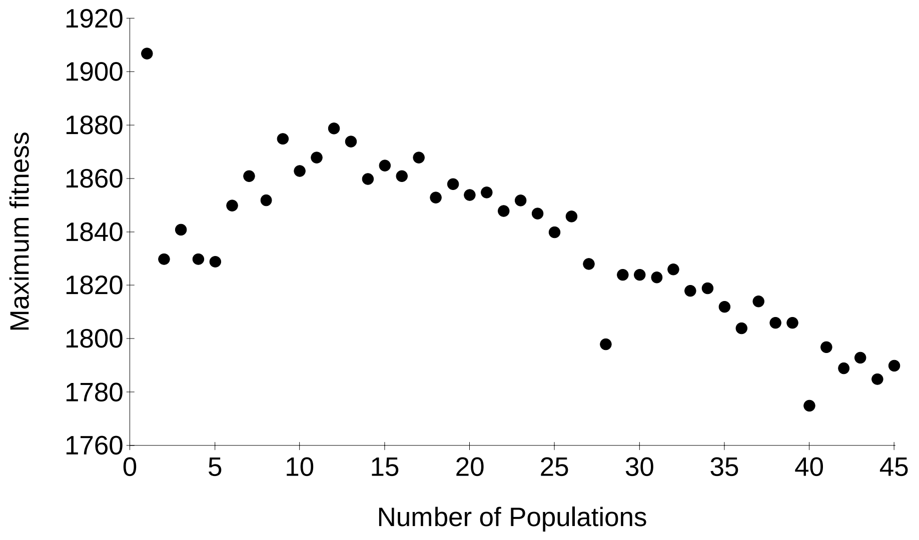


Figure 9
Maximum fitness value attained after 200,000 generations.

The final graph demonstrates that our maximum fitness value does suffer as we increase the number of populations. The best fitness was attained using just one population. It might be possible to improve fitness results by increasing the frequency and degree of migration. These fitness findings are on a very trivial problem and may not hold for more interesting problems.

8. Conclusions & Future Work

As we have seen, the parallel island model of GAs is a very good candidate for use with a PVM network. From humble beginnings, we developed a PVM based island model software solution based on minimizing communication overhead by utilizing fewer messages and avoiding synchronization (except at shutdown time).

Our results show when compared to a single population serial GA, our software attains a sub-optimal, but linear speedup. When compared against a serial island model GA, our software attains near optimal linear speedup.

In the future, we'd like to extend our results to include a more diverse mix of processing elements. All workstations in this paper were of the same computational power, it would interesting to study the effects of mixing fast and slow processing elements together.

The fitness performance of the software also needs a more in-depth study using non-trivial problems. The degree and frequency of migration could be varied to help improve performance. Another interesting idea would be to vary the types and parameters of the GAs running on each machine. This might promote diversity and assist in convergence to the global maximum.

References

[1] A. Corcoran. *An Overview of Parallelism in Genetic Algorithms*. Technical Report, The University of Tulsa, 1993.

[2] A. Corcoran and R. Wainwright. *A Parallel Island Model Genetic Algorithm for the Multiprocessor Scheduling Problem*. Technical Report, University of Tulsa.

[3] W. Erben and J. Keppler. *A Genetic Algorithm Solving a Weekly Course-Timetabling Problem*. Practice and Theory of Automated Timetabling, First International Conference, 1996.

[4] C. Darwin. *On the Origin of Species by Means of Natural Selection*. Cambridge, London, 1964. Harvard University Press (Reprint), Stuttgard, 1906 : Alfred Knoner Verlag (German).

[5] K. De Jong and J. Sarma. *On Decentralizing Selection Algorithms*. Genetic Algorithms: Proceedings of the 6th International Conference, Morgan Kaufmann, San Francisco, CA.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

[7] J.H. Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, University of Michigan Press, 1975.

[8] P. Husbands. *Genetic Algorithms for Scheduling*. AISB Quarterly, No. 89.

[9] G. Lin and X. Yao. *Parallel Genetic Algorithm on PVM*. Proceedings of the International Conference on Parallel Algorithms, Wuhan, P.R. China.

[10] M. Wall. *GAlib: A C++ Library of Genetic Algorithm Components.* Massachusetts Institute of Technology, 1996.

[11] D. Whitely. *Cellular genetic algorithms*. Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, California, 1993.