A Parallel Approach to Improving the Evolution Mechanism in Genetic Algorithms

Asanka Herath and Buddhika Kottahachchi
{asanka, buddhika}@mit.edu

Wednesday, May 12, 2004

Introduction
Genetic Algorithms are intrinsically tied to randomness. Since such algorithms usually do not have the ability to construct good genomes, a considerable amount of time is spent evolving the gene pool before a desirable solution can be found. In this project, we explored a novel approach to addressing this problem by creating good genes in parallel and injecting them to an existing and evolving gene pool, thereby helping that gene pool improve rapidly.

Genetic Algorithms
Genetic Algorithms[1,2] were formally introduced in the 1970s by John Holland of the University of Michigan. They essentially attempt to emulate evolution in biological systems to computing solutions for optimization problems. Initially a gene pool is seeded with candidate solutions (either randomly created, or found by other means involving knowledge of the nature of the problem). Thereafter, a process of evolution involving iterations that apply operations such as crossover and mutation to create new generations of the gene pool. The selection process is biased towards genomes with higher fitness functions moving on to the next generation. Thus, as we iterate through generations the genomes converge towards the optimal solution for that problem.

Parallelized Approaches to Genetic Algorithms
The performance of genetic algorithms with respect to the solutions they generate vary based on the population size considered and the amount of variance allowed. Thus, a larger population allowing a high degree of variation is more likely to produce an individual genome with a high fitness value (that may be your optimal solution) faster than that considering a smaller population. Previous parallel approaches to this problem have focused on maintaining and iterating through larger gene pools as allowed by the availability of more computation power [2]. While it is generally accepted that maintaining larger populations do provide better solutions, we decided to take an alternative approach.

Parallel Genetic Algorithms with Fitness Injection
We chose to focus on an approach that applies parallelism to maintain both a large population, and to expedite the convergence process by injecting high quality genomes generated using a different mechanism into the population.

In particular, we do the following:

1. Devote a set of nodes running a genetic algorithm towards maintaining independent populations of genomes (this approach falls in to the general category of coarse-grained parallel genetic algorithms).
2. Augment this by devoting a second set of nodes running a faster independent algorithm working on the same problem.
3. Share good solutions generated in the second set of nodes by injecting them as genomes in to the individual populations running genetic algorithms.
4. After the populations have evolved for a stipulated number of generations, extract the best genome from the gene pool and present it as an optimal candidate.

Why would this work?
As alluded to above, genetic algorithms are heavily dependent on randomness to improve the quality of their genome populations. Thus evolving to a desirable solution takes time. Furthermore, evolving from a genome with a "decent" fitness value to one with a better fitness value may involve many intermediate generations with the fitness value perhaps dropping very low before it returns to the higher level (Figure 1).

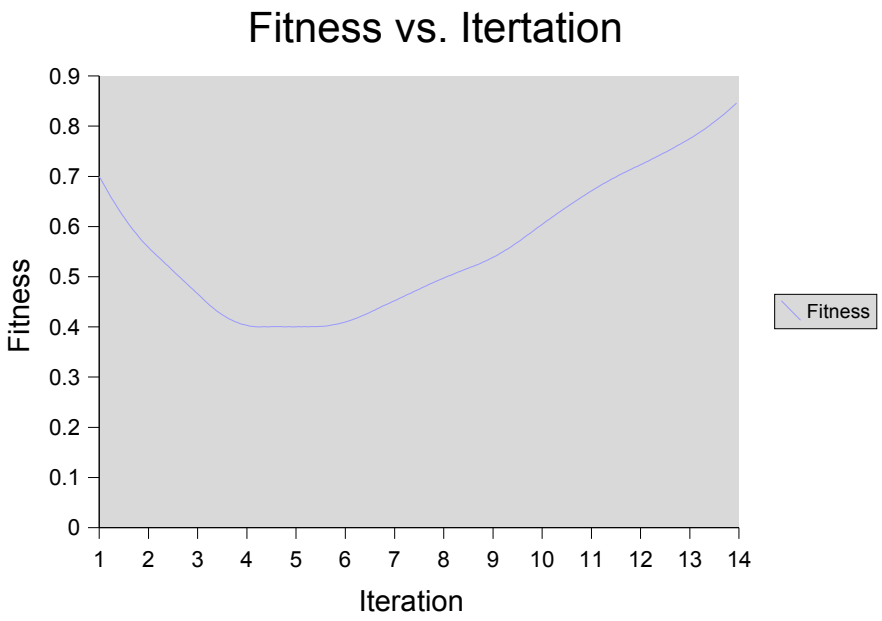

Figure 1: fitness value plotted against iterations of a GA

However, if the independent mechanism generated a genome that was equivalent to that on one of the later iterations shown on Figure 1, then our gene pool improves faster than via the traditional approach. The key is being able to inject an appropriate number of genomes without biasing the end solution towards a local optimum.

Test Domain: Bin Packing Rectangles

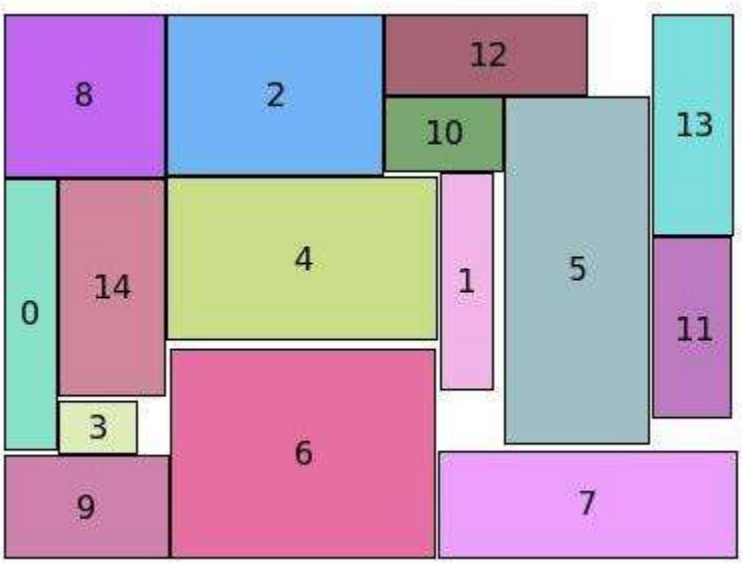

Figure 2: A bin packed with rectangles
(from LinPacker website[http://freehackers.org/~tnagy/])

We chose to apply our approach and test our hypothesis on a variation of the Bin Packing problem. The problem we chose to address is defined simply as follows:

> *Given a set of rectangles and a bin width, pack the rectangles into a bin such that it has minimal height. Rectangles can only be placed on Horizontal or Vertical orientations.*

There exists a reasonably large body of research that has been conducted with respect to this problem. Our approach was tested against data generated by the authors of [6].

Implementation Architecture and Design

Our architecture consists of 3 different types of roles assumed by the nodes. They are described below.

Slave Node – Fast Simulated Annealing
For our candidate genome generator, we selected a fast simulated annealing implementation for the bin packing problem. Nodes assuming the role of a slave running simulated annealing work on the problem independently to generate candidate solutions. The implementation is based on the LinPacker[4] library which is available under the GNU Public License[5]. Since this works faster than the genetic algorithm, for each iteration on the nodes running the genetic algorithm these nodes run n iterations (was tested for n=50). Upon completing a batch of iterations, the existing best solution is sent to the Master Node. Thereafter, the next batch of iterations is run. It is important to note

that state is maintained, and thus candidate solutions generally improve between successive batches of iterations.

Slave Node – Genetic Algorithm
Nodes running the genetic algorithm work on an independent population of genomes. The particular implementation is also based on the LinPacker library. At the completion of each iteration these nodes pull down a set of candidate genomes generated on the pool of slave nodes running the simulated annealing algorithm. The worst genome available locally is replaced if the one received has a higher fitness value. This is the process we refer to as injecting solutions. Furthermore, we implemented a mechanism which reduces the likelihood of a genome being injected in if it has been injected once. This is to prevent a bias towards a local optimum.

Master node
This node handles all co-ordination tasks related to solution generation. For example, this node performs the following tasks:
1. Read in test cases and preferences
2. Push out test cases and preferences to the slaves
3. Retrieve candidate genomes from slaves running the fast simulated annealing approach.
4. Push out candidate genomes to slaves running the genetic algorithm.
5. Once the number of iterations are completed, pull the best solutions from the slaves running the genetic algorithm, select the best one and write it out as an optimal candidate.

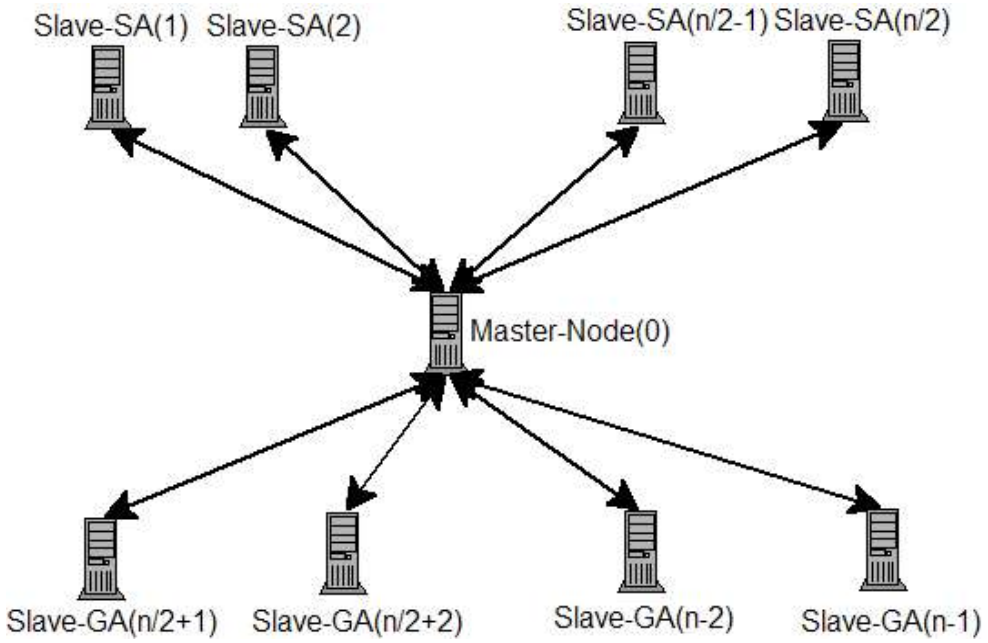

Figure 3: Software Architecture

The code for the nodes are implemented using C++ and we use MPI to enable inter-node communication. All nodes essentially run the same body of code, but use knowledge of

their local node ID to assume a particular role. The first node is immediately assigned as the Master, while the next $\lfloor n/2 \rfloor$ are assigned the role of Slaves running simulated annealing and the remaining nodes run the genetic algorithm.

Validation Case
The test case used for the hypothesis validation runs were extracted from [6]. The test case used had the following characteristics.

- 60 Rectangles
- Each side has a maximum length of 100 units
- Bin width is set to 100 units
- The total area covered by the rectangles is 210138 $unit^2$
- Thus the perfect ideal solution has a height of about 2102 units
  - We cannot guarantee that such a solution exists

A slim bin width was selected to enable a higher variance in the solution heights such that any improvements would be noticeable.

Testing
We ran two comprehensive sets of tests on this validation case. The first involved 50 iterations of the genetic algorithm while the other performed 100 iterations. Each set of tests involved a control case and the test case. They are defined as follows:

Control Case: This case has all the nodes running the genetic algorithm on independent populations with no solution injection taking place.

Validation Case: This case has the nodes being allocated tasks as described above in the implementation architecture and design section.

All tests were run on 16 virtual nodes. This meant that for the validation case we had 1 master node, 8 slaves running simulated annealing and 7 running the genetic algorithm. In the test case, because of constraints in the way our code was structured, we had 15 nodes running the genetic algorithm.

Results and Interpretation
Both sets of tests for the two cases described above were performed 10 times each. Since the algorithm's are inherently random, we needed a sufficiently large set of results of who's average value we could use to test our hypothesis.

Our investigation yielded the following results:

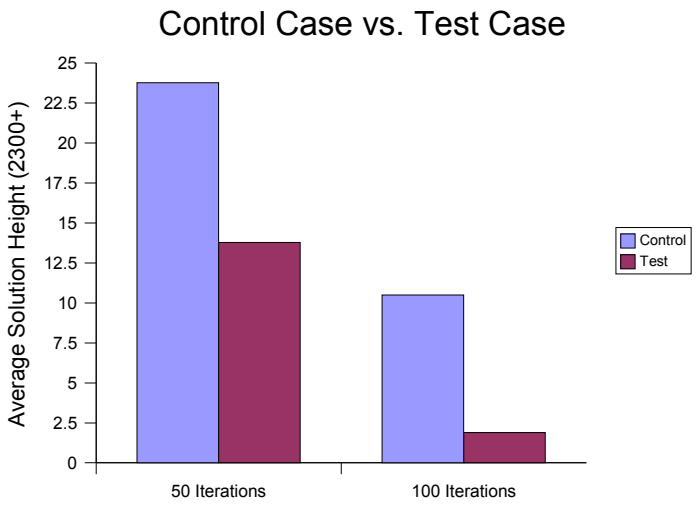

Figure 4: Validation Testing Results

As can be seen, on average the test case which was running our approach performed better on both the 50 and 100 iteration tests. It can be argued that the improvements are marginal. However, we argue that even though they are marginal they are significant. To begin with, the solutions generated by the algorithms running on our slave nodes were of very high quality. Quality was measured as the wastage resulting from a packing (we observed wastage to be around 10-12% in the solutions generated by our implementations). Thus, marginal improvements required vast amounts of computation. The fact that our approach consistently generated better solutions than the control case adds credibility to the validity of our hypothesis.

Conclusions and Future Work

Based on our preliminary testing, we believe that our approach merits further investigation. We understand that the problem we chose, was perhaps non-ideal for testing our hypothesis considering it arrived at high quality solutions very quickly. Therefore, exploring this approach on other problem domains would be desirable. Also a means for tracking the quality of the genomes through generations would further help test our hypothesis.

References:

[1] John H. Holland: Genetic Algorithms and the Optimal Allocation of Trials. SIAM J. Comput. 2(2): 88-105 (1973)
[2] J.H. Holland, “Adaptation in Natural and Artificial Systems”, MIT Press, 1975.
[3] Erick Cantu-Paz, “A Survey of Parallel Genetic Algorithms”, IlliGAL Report, 1997.
[4] http://freehackers.org/~tnagy/
[5] http://www.gnu.org/licenses/licenses.html
[6] Hopper E. and Turton B. C. H., 2002, "An empirical study of meta-heuristics applied to 2D rectangular bin packing" Special Issue on Cutting, Packing and Knapsacking Problems, Studia Informatica, vol. 2, no. 1