A Generalized Parallel Genetic Algorithm in Erlang

Amanda Bienz
Elon University
Elon, NC
abienz@elon.edu

Kossi Fokle
IUPUI
Indianapolis, IN
ktfokle@iupui.edu

Zachary Keller
DePauw University
Greencastle, IN
zacharykeller_2012@depauw.edu

Ed Zulkoski
Wilkes University
Wilkes-Barre, PA
edward.zulkoski@wilkes.edu

Scott Thede*
DePauw University
Greencastle, IN
sthede@depauw.edu

ABSTRACT

The focus of this paper is to implement a genetic algorithm using parallel programming. Genetic algorithms are well-suited to "parallelization," since they model many individuals. Three implementations of a genetic algorithm were created for this paper - a standard sequential programming algorithm, a parallel algorithm using a master process to control the algorithm's operations, and a parallel algorithm using a grid structure for the individuals. These implementations were tested on a single workstation as well as a server with many processors. The parallel algorithms outperformed the sequential algorithm, and their performance improved when run on the server with more processors.

1. INTRODUCTION

A genetic algorithm, at its essence, finds a solution to a problem. Genetic algorithms can be particularly useful on problems with a prohibitively large number of possible solutions. While a genetic algorithm may not be guaranteed to find the "best" answer, it provides a good solution in a reasonable amount of time. Due to this applicability to intractable problems, genetic algorithms have found use in a wide variety of fields, from biology [10] to graph theory [2] to industrial engineering [6]. Given the widespread utility of genetic algorithms, we decided to investigate ways to improve their speed and performance.

The focus of this paper is to describe the use of parallel programming to implement the genetic algorithm. Intractable problems frequently have an incredibly large number of possible solutions, and the only way to determine the best solution is to consider them all. A genetic algorithm cannot look at all the possible solutions, but the more it can consider, the better its results will be. By implementing a genetic algorithm using parallel programming, we hope that it will consider many more solutions in a short period of time, thus increasing the quality of the solution it eventually finds.

Since genetic algorithms can be used in so many fields, we feel that it would be useful to have a generic genetic algorithm; in other words, a genetic algorithm that could be applied to any problem. Similarly, genetic algorithms have quite a few independent parameters that are set by the programmers. For example, the rate at which mutations occur, or the method in which fitness is calculated, are somewhat arbitrary and possibly problem dependent. We will consider the variation in many of these parameters and attempt to optimize their values for a generic genetic algorithm.

In summary, in this paper, we hope to accomplish three things:

1. Implement a genetic algorithm using parallel programming.
2. Perform this implementation in such a way as to be as problem-independent as possible.
3. Optimize some of the parameters of the algorithm to maximize performance.

*Faculty advisor

Proceedings of the 2011 Midstates Conference on Undergraduate Research in Computer Science and Mathematics

2. BACKGROUND

A genetic algorithm is a search algorithm that finds as good a solution as possible to a problem by emulating the process of (biological) evolution. While there is no guarantee that the optimal solution will be found, the genetic algorithm can generally find good solutions relatively quickly. A genetic algorithm is based on the concept of "survival of the fittest." In the biological world, people, animals, and other organisms compete among themselves on a daily basis to gain both resources vital for survival and suitable mates. Successful individuals are more likely to survive and are capable of producing the most offspring. As a result, the majority of new generations should contain mainly fit individuals and their offspring [1].

A genetic algorithm models a population of individuals, as in biology, but each individual represents a potential solution to the problem the algorithm is trying to solve. There are a number of concepts used in the genetic algorithm to model the survival of the fittest mechanic.

Fitness The "fitness" of an individual is based on how good the solution is to the problem in question. Individuals that are more fit are more likely to reproduce, while unfit individuals are more likely to die out. However, it is still possible for poor solutions to survive and reproduce [7].

Cross-Over Also known as "breeding," this is the mechanism by which individual solutions create a new individual. In the case of a genetic algorithm, two individuals are selected (in such a way that more fit individuals are more likely to be selected) and their attributes are combined to form a new individual. The way this combination is achieved is generally problem-dependent, but can be generalized depending on how the individuals are represented.

Mutation Each newly created individual in the population has a chance of mutating. In this case, this means making a relatively minor change to the attributes of the individual. Mutation allows the genetic algorithm to explore more options, and avoid becoming "trapped" in a local maximum.

Given these operators, a genetic algorithm generally operates in the following fashion:

1. Create an initial population of individuals randomly.
2. Set other initial parameters to appropriate values.
3. Repeat until algorithm is complete:
   (a) Select 2 individuals (so that fitter individuals are more likely to be selected).
   (b) Breed, or cross-over, those two individuals, creating a new individual.
   (c) Possibly mutate the new individual.
   (d) Add the new individual to the population, possibly removing one or both of the parents.
4. Report the best individual from the final population.

The algorithm is complete when a preset condition (determined when starting the algorithm) is met. For example, the algorithm could be set to run a certain number of seconds, or it could be set to run for a certain number of generations, or it could be set to stop when a certain number of generations pass without improvement in the best individual.

3. DETAILS ON GENETIC OPERATORS

The two types of genetic operators used are mutation and crossover. We will discuss implementation-specific details of each here.

3.1 Mutation

Mutation alters the binary representation of one individual, resulting in a new solution [7]. There are multiple types of mutation, including binary mutation and mutation of a permutation.

Binary mutation converts each element in the individual to binary digits. Mutation then assigns a certain probability that each binary digit will "flip."

For example, a 16-bit individual encoded as 1101011100011011 might have a mutation in the seventh bit, and become the new individual 1101010100011011. The mutation rate is typically set at a level where it is unlikely for many bits in an individual to change.

For a permutation problem, such as the traveling salesman problem, mutation should only change the order of an individual's elements. Using the "bit flipping" method could result in an invalid solution to the problem, since the resulting individual

must be a permutation of the original one [9]. Mutation in a permutation problem selects on element with a specified probability and randomly selects another element. The two selected elements swap location, resulting in a new permutation.

For example, an individual encoded as the list [7,4,5,3,1,6,2] might mutate by randomly selecting the 4 and the 1, resulting in the new individual [7,1,5,3,4,6,2]. This preserves the individual as a valid permutation while changing it slightly from its pre-mutation form.

3.2 Crossover

Crossover takes two different individuals and swaps some elements between the two, creating two new solutions [7]. There are many approaches to this, including simple crossover at the binary level, real valued simple crossover, and crossover of two permutations.

Simple crossover at the binary level converts two entire individuals to binary. A random position is chosen, and both binary representations are split at this position. The two individuals combine at this splitting point, and then convert back from the binary values.

For example, two eight-bit individuals, 11010111 and 01100001, might be selected to crossover, with the crossover point randomly selected as between the third and fourth bits. The two children that could result from this crossover would then be 11000001 (the 110 coming from the first individual and the 00001 from the second) and 01110111 (the 011 from the first and the 10111 from the second).

Real valued simple crossover takes splits the elements of two individuals at a random position. The real valued elements before the split of one individual are combined with those after the split of the other individual, and vice versa. This works just like the simple crossover described above, except with real valued elements.

The results of a crossover of two permutations must also be two permutations of the same elements. While the order can change, the elements in the permutations must be the same. To do this, a group of elements in each individual can be selected. These elements are then reordered to mimic the order of the other individual in crossover, resulting in two new permutations of the problem.

For example, say we have two individuals represented as [5, 2, 3, 1, 4] and [4, 3, 1, 5, 2]. Assume a crossover point is given as between the second and third number in the list. We cannot simple swap the values, as we did above, because then the solutions would not be valid permutations. Instead, we consider the second region of the first individual (the 3, 1, and 4). We keep these elements in that region, but we reorder them to match the order in the second individual (4, 3, 1). Thus, the first possible child is [5, 2, 4, 3, 1]. Similarly, the second possible child is [4, 3, 5, 2, 1]. This gives us new children that contain information from both parents.

4. THE ALGORITHMS

Three different implementations of a genetic algorithm are discussed in this paper. Those three implementations are:

1. A sequential algorithm
2. A parallel algorithm using a master process approach
3. A parallel algorithm using a grid approach

These algorithms were implemented in Erlang. This programming language choice is discussed briefly, then the three implementations are given more detail in this section.

4.1 Erlang

We chose Erlang as the programming language to use for the work done in this paper. Erlang was developed in the early 1990s at Ericsson, for use in the telecom industry. Important features of Erlang with respect to this paper are its use of concurrent processes, message passing, and scalability in parallel problems [3].

Erlang was chosen for this work because it would allow us to model each individual in the population of the genetic algorithm as a single process. Given the size of populations in a typical genetic algorithm (we used 10,000 individuals when testing our algorithms), this requires a language that can handle large numbers of concurrent processes at once. The use of message passing as a communication method between processes fits in well with our algorithms' approaches (see below).

4.2 Sequential Approach

The sequential algorithm implements a "classic" genetic algorithm, and was intended as the "control" for this experiment. The pseudocode for a typical genetic algorithm was followed. The population is initially created randomly. The fitness for each individual in the population is then calculated. Based on their fitnesses, certain individuals are selected to breed and create offspring. The new offspring have a certain probability of mutating. These offspring create a new population of individuals. The process is repeated until the algorithm finishes (based on total running time or total generations).

4.3 Master Process Approach

This approach treats every individual in the population as a process. These individuals are organized by another process, known as the master process. This master process controls how the individuals in the population interact. Since all processes need to communicate through it, the master process should do as little work as possible in order to minimize the bottlenecking which could occur. Our algorithm has individuals notify the master process after they have completed their fitness calculations. The master process then groups individuals together into groups of arbitrary size. Our experiments used groups of size four, to maintain similarity to the grid approach (described below). Once the four individuals have been grouped together, one of those individuals is designated as the group leader.

The other processes of the genetic algorithm - selection, crossover, and mutation - are performed within the small group, independent of the master process. Members send their fitness values and chromosome to the leader process, which then performs all functions. The members of the group are modified to become their offspring as necessary, then they reevaluate their fitness function and report to the master process again.

This approach uses one process as the master process, plus a single process for each individual in the population. Any size population can be modeled with this approach; we tested it on populations ranging from a few individuals up to 100,000 individuals with no problems. The data reported in this paper used a population size of 10,000.

4.4 Grid Approach

Parallel programs can be run on anywhere from one single processor to hundreds of processors at once. As the number of processors increase, a larger number of threads can run simultaneously. When there is a large number of threads running, a central process controlling them could slow the program down.

Our second parallel approach has each individual process perform its own selection. This avoids the possibility of a bottleneck occurring in a single master process. However, each process needs to communicate with the other processes in the population. If there are a large number of processes, and every process must communicate with every other process, the computer could run out of memory, and the program could bog down.

To avoid this, imagine all of the processes being stored in a two dimensional array, or grid. Each process can communicate with every neighboring process, particularly those directly above, below, and to each side.

To perform selection, each process sends its fitness value to every neighboring process, then waits to receive either a specified number of communications from neighboring individuals, or a new individual to replace itself. If a process receives information from neighboring individuals, both selection and breeding are done without the knowledge of any other processes. The resulting children are sent to their respective parents. These children replace the parents, then a new generation begins.

This approach avoids the need for a process as a "master" process, and thus uses a single process for each individual. Again, we tested on sizes of populations up to 100,000 with no problems. The data reported later in the paper used a population size of 10,000.

5. THE TEST PROBLEMS

We chose to test our genetic algorithms on two problems - the Traveling Salesman problem (TSP) and the Bin Packing problem (BPP). These problems were selected for a number of reasons. They are "classic" problems in computer science. They are both NP-complete problems, which means that they are extremely unlikely to ever be solved by a polynomial running time algorithm. This means that they are well-suited to being attacked by non-brute force methods, such as a genetic algorithm.

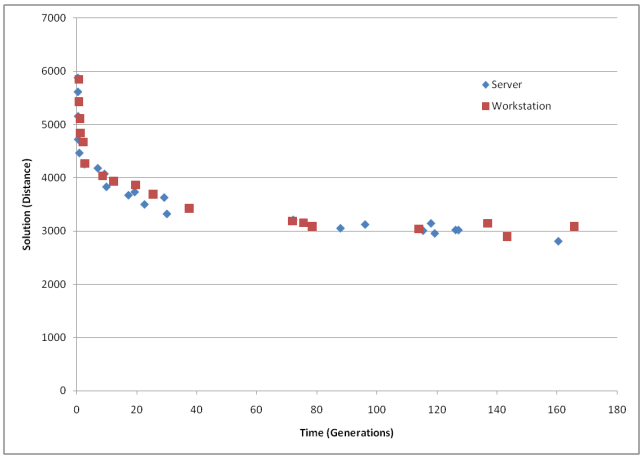

Figure 1: TSP Using Sequential Algorithm

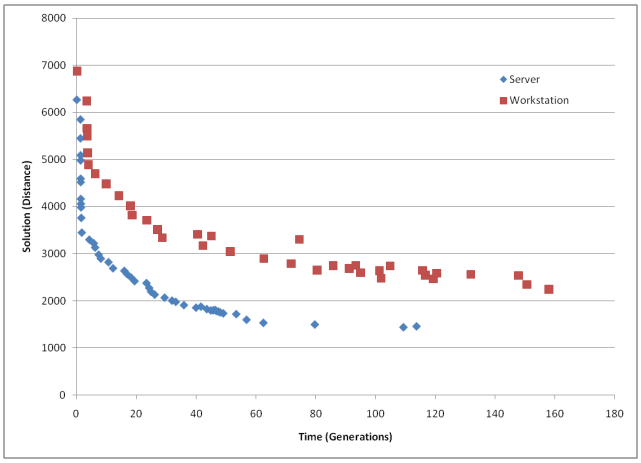

Figure 2: TSP Using Grid Algorithm

Additionally, while they are both NP-complete, they are fairly different in their specifications. Using two different problems for testing allows us to make sure the algorithm is general enough to handle both. We describe each of the problems briefly here.

5.1 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) states that a traveling salesman has some set of cities that he services, including his home city. The salesman must travel from his home city to every other city and return home. He must visit each city only once (excepting the fact that he starts and finishes at his home city). He also wants to minimize the cost of his trip. The problem requires us to find the minimum cost trip that leaves from the home city, visits every other city once, and returns home.

This relatively informal description can be translated to a graph theory problem. Given a graph with a set of N vertices and a set of weighted edges connecting these vertices, find the minimum cost cycle in this graph that starts and ends at a specific vertex V and visits every other vertex only once. Thus, the cycle consists of exactly N edges. In practice, the requirement to start and stop at a particular vertex is not necessary, as long as the cycle visits every vertex. This problem has been proven to be NP-complete [5]. To find the exact minimum solution, it is necessary to consider all possible paths, which requires a running time of O((N-1)!), where N is the number of cities.

5.2 Bin Packing Problem

The bin packing problem (BPP) states that a company has a number of bins, each with a particular

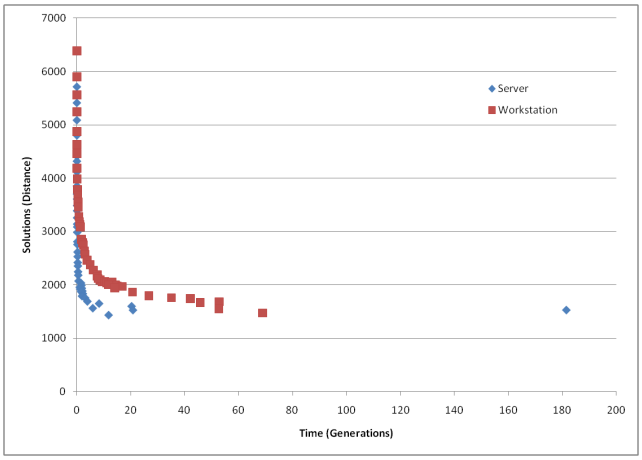

Figure 3: TSP Using Master Algorithm

capacity, and a large collection of items, all of different sizes. The problem asks if there is a way to assign all the items to a bin such that every item is in a bin, and no bin exceeds its capacity. In more formal language, given a finite set U of items, a size(u) function defined for all elements of U such that size(u) is a real number larger than 0, a bin capacity B, and a number of bins K, can U be partitioned into K subsets so that the sum of the sizes of the items in each subset is less than or equal to B. This problem has also been shown to be NP-complete [5].

6. RESULTS

The results of the algorithms on the two problems are given in this section. Each algorithm was tested on two machines, a sixteen-core server and a four-core workstation.

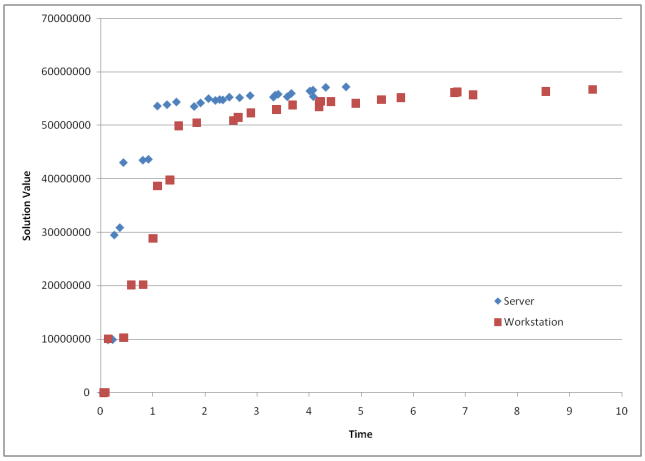

Figure 4: BPP Using Sequential Algorithm

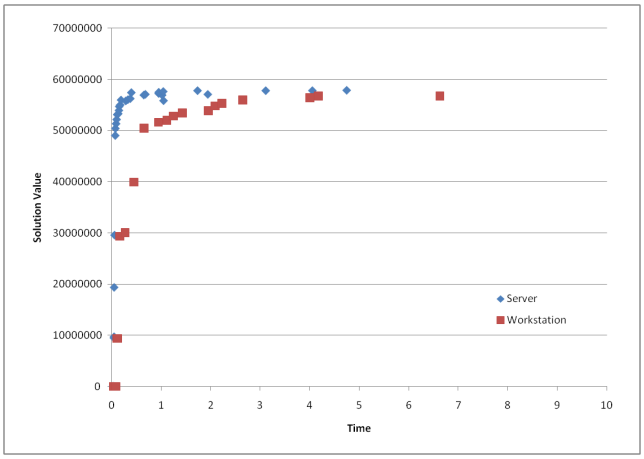

Figure 5: BPP Using Grid Algorithm

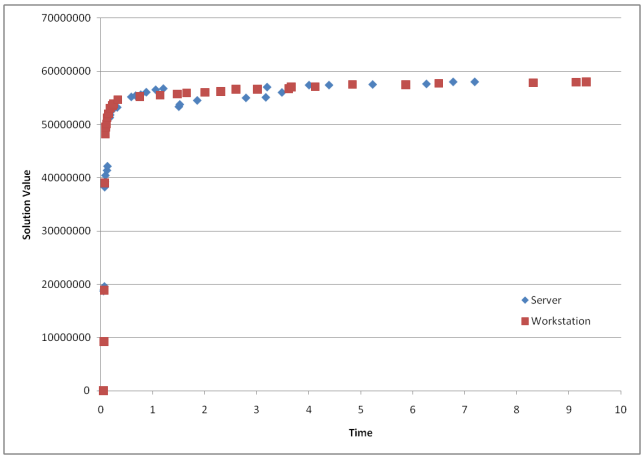

Figure 6: BPP Using Master Algorithm

6.1 TSP Results

These three graphs (Figures 1, 2, and 3) plot the total distance of the best solution in the population versus the time (measured in generations). Therefore, a smaller solution value is a better fitness for this problem. As can be seen, there is a large initial improvement in the quality of the solution in all algorithms, followed by a modest improvement over a longer time frame. As expected, the machine running the sequential algorithm had little effect on the algorithm's performance, since the algorithm runs on a single processor. The sequential algorithm also provided the worst solutions, eventually reaching a best solution with a distance of 2809.

The parallel algorithms each outperformed the sequential algorithm, with both eventually finding solutions with a distance of approximately 1400 (the grid algorithm finds a solution of 1433, and the master process algorithm finds one with 1430). Notice that the master process algorithm reaches its best performance much more quickly than the grid algorithm, particularly on the server. In both parallel algorithms, running them on the server (with more processors) offers better performance and a quicker convergence to a good solution.

Note that the time is measured in generations, not in actual units of time. This allows us to avoid issues with timing algorithms, such as differences in implementation as well as differences in machine load at the time of testing. Genetic algorithms tend to improve the longer they are run - there is no real pre-defined "end point." Therefore, plotting algorithm performance vs. generation allows us to more easily compare performance by the different algorithms. For example, note that each parallel algorithm surpassed the best solution found by the sequential algorithm within 10 generations, and the final results were less than half the size of the solution found by the sequential algorithm.

6.2 BPP Results

Performance of the three algorithms on the BPP is a little more varied. In this problem, the solution value is measured using a function based on how evenly the bins are packed. This function is constructed so that a larger value means more evenly packed bins, so larger valued solutions are better fitnesses.

In the end, each of the algorithms finds solutions of similar value (as shown in Figures 4, 5, and 6), although it takes the sequential algorithm a bit longer to reach it. Our belief is that the prob-

lems we tested on were somewhat too simple, allowing each algorithm to converge to an optimal or near-optimal solution in the time frame given. It is somewhat puzzling that the sequential algorithm seems to perform better when run on the server than the workstation, although this might be due to a difference in speed between single CPUs on those machines.

We are also somewhat puzzled why the performance of the master process algorithm seems to be roughly identical on the server and the workstation. This may be due to the algorithm finding the solution so quickly that differences between the machines become negligible. After all, the grid algorithm seems to perform much faster on the server. We should also point out that there is a certain amount of randomness inherent in genetic algorithms, so minor variations can be expected.

7. FUTURE WORK

There are many plans to continue the work described in this paper. We would like to further test the generic nature of the algorithm by testing on more problems. We would also like to more rigorously test the significance of the number of processors used on the machine to run the algorithm. Particularly, would the performance be significantly improved by running the algorithms on machines with massive numbers of CPUs (on the order of 1 CPU per individual)? Finally, there are many parameters used in the algorithms (such as the number of process in a group in the master algorithm, the mutation rate, the number of threads to allocate to the algorithm, and so on). The differences in performance that would result from changing these parameters could be more rigorously studied as well.

8. CONCLUSION

In conclusion, we have created two algorithms to execute a genetic algorithm using parallel processing. These two algorithms have been shown to perform faster and find better solutions than a standard sequential algorithm. There is also evidence to show that increasing the number of processors in a machine increases the performance of the parallel algorithms.

9. ACKNOWLEDGEMENTS

Work on this project was funded by the National Science Foundation, through a Research Experience for Undergraduates grant, REU grant CFF-0851812.

10. REFERENCES

[1] Beasley, D. An Overview of Genetic Algorithms: Part 1, Fundamentals. University Computing. 1993.

[2] Bui, T. and Moon, B. Genetic Algorithm and Graph Partitioning. IEEE Transactions on Computers 45(7): 841-855. July 1996.

[3] Cesarini, F. and Thompson, S. Erlang Programming. O'Reilly Publishing. 2009.

[4] Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers. 2000.

[5] Garey, M. and Johnson, D. Computers and Intractability: A Guide to the Theory of NP-completeness. W.H. Freeman and Company. 1979.

[6] Hopper, E. and Turton, B. A Genetic Algorithm for a 2D Industrial Packing Problem. Computers and Industrial Engineering 37(2): 375-378. 1999.

[7] Houck, C., Joines, J., Kay, M. A Genetic Algorithm for Function Optimisation: A Matlab Implementation. North Carolina State University. 1996.

[8] Muhlenbein, H. Evolution in Time and Space - The Parallel Genetic Algorithm. Foundations of Genetic Algorithms. Morgan Kaufmann. 1991.

[9] Serpell, M. and Smith, J. Self-Adaptation of Mutation Operator and Probability for Permutation Representations in Genetic Algorithms. Evolutionary Computation 18(3): 491-514. Massachusetts Institute of Technology, 2010.

[10] Shyum C., Sheneman, L., Foster, J. Multiple Sequence Alignment with Evolutionary Computation. Genetic Programming and Evolvable Machines 5: 121-144. 2004.

[11] Zorman, B., Kapfhammer, G., and Roos, R. Creation and Analysis of a Javaspace-based Genetic Algorithm. Proceedings of the 8th International Conference on Parallel and Distributed Processing Techniques and Applications. June, 2002.