YALE UNIVERSITY

Using a Cellular Genetic Algorithm with MPI to Approximate the Minimum Vertex Coloring Problem

Tyler Petrochko

May 10, 2017

1 BACKGROUND

The Minimum Vertex Cover Problem (MVCP) belongs to the set of problems classified as NP-Hard. In essence, NP-Hard problems can be reduced to or from any other NP-Hard problem, and cannot be solved in polynomial time (unless P=NP). However, this only applies to the best solution; in practice, a close-enough approximation can often be found in polynomial or even linear time.

The goal of MVCP is to find a vertex cover for a given graph $G = (V, E)$ of least size, where a vertex cover V' is defined as a subset of V where every edge in E touches at least one vertex in V'. For a graph with no edges, the empty set is a vertex cover, and for any graph, V itself is a vertex cover. Solving this problem via brute force would require checking every single subset of V, an operation that would take $O(2^n)$ time complexity. However, better methods exist.

The most naive approach to approximating MVCP is as follows: keep selecting the vertex of highest degree in G, place it in V', then remove all incident edges to the vertex. This algorithm tries to cover the most ground (in terms of edge count) with every vertex that is removed, but as demonstrated by Papadimitriou and Steiglitz (professors at Berkley and Princeton respectively), this approach falls short in many circumstances. A better approach works randomly choosing an edge (u, v) in E, adding both vertices to V', then removing all edges incident to either vertex.

One alternative approach to approximating MVCP requires a genetic algorithm, which searches for a solution via evolutionary concepts such as mutation, reproduction, and natural selection. To do so efficiently, it would be ideal to break the problem into many smaller chunks that can be solved in parallel. We discuss techniques to do this shortly.

Genetic algorithms are stochastic search techniques that start with a set of randomized solutions to a given problem, and form new solutions based of mutations and genetic crossovers. In order to drive this process towards better solutions, a fitness function must be devised that can remove inferior solutions from future generations. The fitness function takes in a numerical representation of a chromosome (usually a binary string of fixed length) and outputs a scalar value to be maximized. A crossover between two chromosomes involves combining the first l genes from a first chromosome with the remaining $n - l$ genes from a second chromosome, where $n < l$, and n represents chromosome length. Mutation functions as a secondary operator, randomly altering certain chromosomes to introduce genetic diversity into evolutionarily stagnant gene pools. This often occurs when a particularly strong solution takes over the rest of the population.

Parallel genetic algorithms divide the simulated evolutionary process into many sub-problems and split work between processors or GPU threads. Parallel genetic algorithms can be one of three types: global single-population master-slave, single-population fine-grained, or multiple-population coarse-grained. The first approach is the most intuitive; a single master worker designates work to a crowd of workers that perform individual fitness evaluations, crossovers, and mutations. In single-population fine-grained parallel GAs, the global population is broken into neighborhoods, with genetic exchange allowed at the borders. This helps prevent genetic stagnation, which we refer to as evolutionary convergence. The last type, multiple-population coarse-grained GAs, attempt to do the same by isolating sub-populations even further, only allowing occasional genetic information to drift between them like ant colonies. In our case, we use a hybrid approach that combines aspects from all three types.

As genetic algorithms advance, they eventually converge upon a single solution, preventing further improvements. Ideally, this convergence will be an optimal solution, but in many cases, the algorithm will get stuck at a local maxima. There are several approaches to prevent convergence, the first being the neighborhood approach mentioned previously.

An alternative design that we explore in this implementation involves arranging cells of chromosomes on the surface of a two-dimensional toroidal mesh. In each iteration of the algorithm, cells are permitted to mate only with adjacent cells, slowing the spread of genetic information and therefore maintaining genetic diversity. As better solutions come up, they slowly effuse throughout the surface of the toroidal mesh, leaving time for alternative approaches to arise. Our implementation combines this approach with master-slave coordination for simplicity.

2 Overview

A two-dimensional toroidal mesh is well-suited for a parallel implementation of a cellular genetic algorithm because it can easily be divided into slices, where each slice is owned by a different processor. Each processor than evaluates fitness, exchanges genetic information with neighboring slices, and performs crossovers or mutations in parallel. The main performance bottleneck for this type of design is the border communication; at the beginning of each iteration, a processor must send a genetic copy of the cells on the left and right edge of its slice to its neighbors, and receive its bordering cells in turn. For simplicity, we make each slice a single cell wide, but the cost of this edge-exchange could be minimized by widening slices and carrying out sub-iterations local to each slice in between edge exchanges. This incorporates elements from the previously-discussed coarse-grained genetic algorithms, and would be much harder to implement but would require less inter-process communication.

While this design does use a master node to coordinate the worker nodes and create the original graph, the implementation could be slightly modified to perform work on the master as well.

Genetic algorithms eventually finish based on one of two criteria: convergence or a fixed-number of iterations. Convergence can be difficult to detect; some implementations forego mutations and stop once the cellular population is homogenous, but this can lead to a sub-optimal solution. For this purpose, our program uses a fixed number of iterations each time (chosen so the simulation continues sufficiently long).

Another design choice to be made is how to carry out mutations. Somewhat counter-intuitively, changing the specific mechanism by which the mutations were performed seemed to have little effect on the genetic algorithm's success; rather, the likelihood of mutation had a much more profound effect, as a too-high mutation rate prevented superior solutions from propagating, but a too-low mutation rate led to early convergence. Two single-point mutations were arbitrarily chosen to achieve this purpose, and the ideal mutation rate is investigated in later sections.

A suitable fitness function must satisfy two conditions: every chromosome representing an infeasible solution must be less fit than every feasible solutions, and chromosomes representing smaller vertex covers should have a higher fitness values than those with large numbers of vertices. For this implementation, we look to a similar implementation:

$$f(\vec{x}) = (n^3 + n) - \sum_{i=1}^{n} (x_i + n \cdot (1 - x_i) \cdot \sum_{j=i}^{n} (1 - x_j) \cdot e_{ij})$$

For any valid vertex cover, the end term will go to zero as no edges are uncovered, making

the fitness at least n^3. For every uncovered edge in an infeasible solution, a penalty of magnitude n is applied to each vertex, leaving a maximum fitness of $n^3 - n$. This satisfies the first condition; the second is ensured by the fact that for any true vertex cover, each vertex diminishes the fitness of the solution by exactly one.

The code for the master worker is as follows:

1. Generate G
2. Distribute G to slaves
3. Wait for slaves to finish
4. Collect all chromosomes from slaves
5. Choose the most fit chromosome as solution

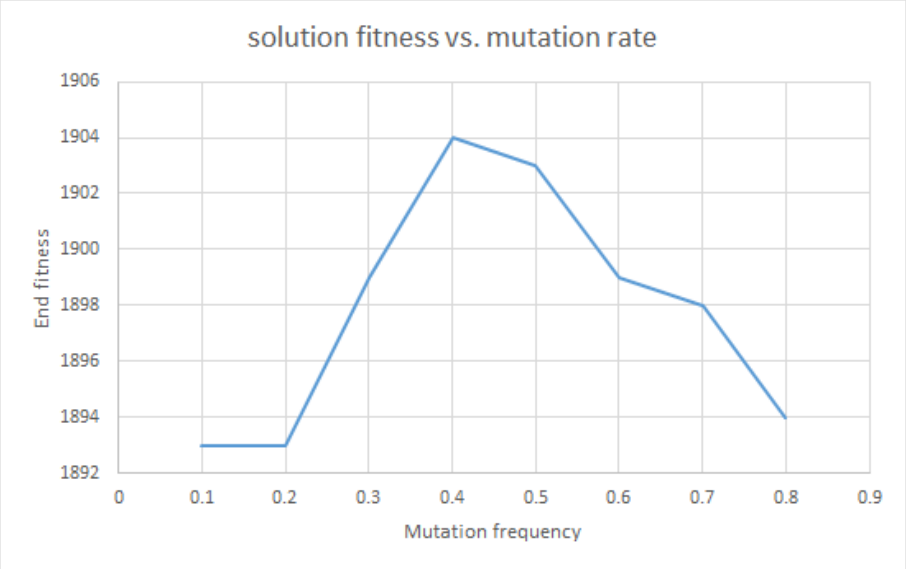
And the code for each slave worker is as the following:

1. Receive G from master
2. For $i = 0$ to $torusHeight - 1$ generate a chromosome (binary string) of length n
3. $numIterations$ times do
    a) If even rank
        i. Send chromosomes to right
        ii. Send chromosomes to left
        iii. Receive chromosomes from right
        iv. Receive chromosomes from left
    b) If odd rank
        i. Receive chromosomes from right
        ii. Receive chromosomes from left
        iii. Send chromosomes to right
        iv. Send chromosomes to left
    c) For $i = 0$ to $torusHeight - 1$
        i. Let $neighbors$ be i's chromosomes to left, right, above, and below
        ii. if i is more fit than all neighbors, maybe mutate it
        iii. else $i = crossover(mostFitNeighbor, secondMostFitNeighbor)$

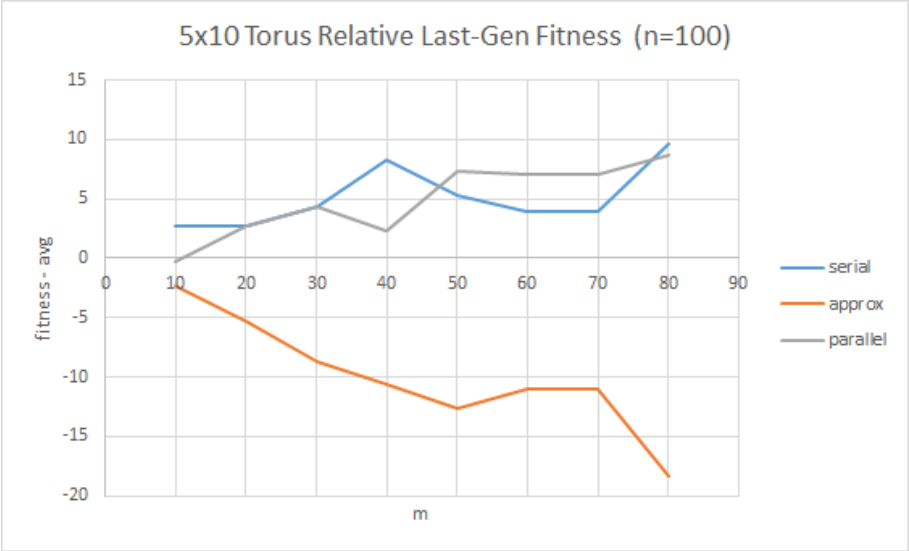
To evaluate our genetic algorithm, we compare the parallelized version with a serial implementation of the same approach, as well as the randomized approximation heuristic outlined earlier. We test a variety of mutation rates and torus dimensions, while keeping the number of iterations constant at 100 for consistency. We then assess the scalability of the parallel version, and the ending fitness of the genetic algorithm as compared to the randomized heuristic.

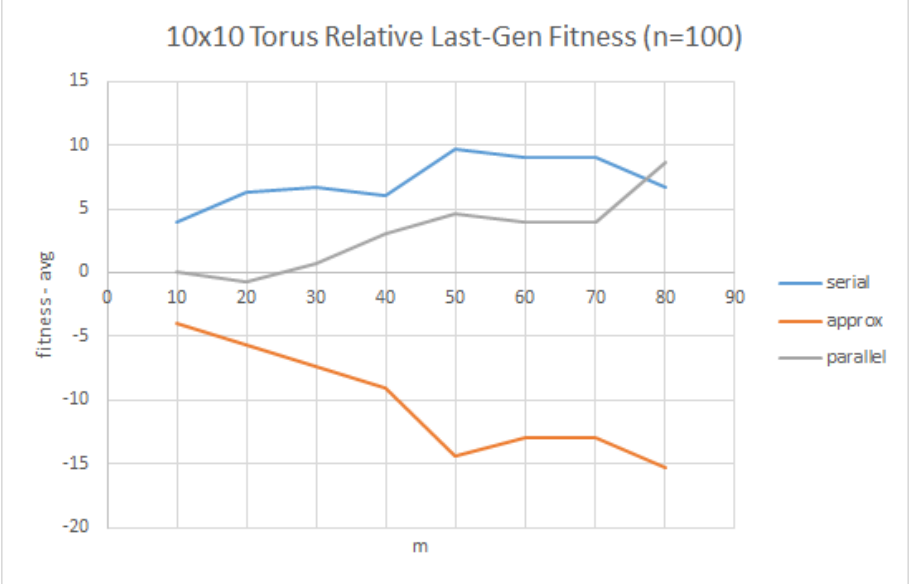
3 Results

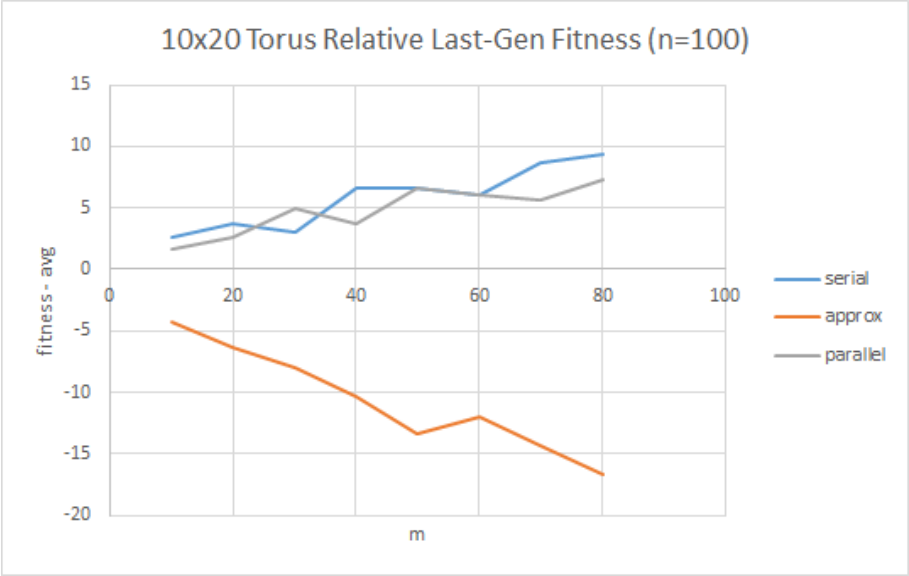
Results First, we attempt to determine the optimal mutation rate. This test was conducted with a torus of dimensions 5x10, but tests on other torus sizes yielded consistent results.



This suggests that four of every ten dominant chromosomes - ones more fit than all their surrounding neighbors - should undergo a two-point mutation. This metric was applied to all future tests. Next, we test whether our genetic algorithm implementation outperforms the randomized heuristic in terms of fitness. We test three torus dimensions: 5x10, 10x10, and 20x10 on graphs of varying edge-density. The following charts portray the last-generation fitness of the serial and parallel implementations compared to the heuristic approach for graphs with a fixed number of vertices and varying edges. Note that that value on the y-axis represents the difference from the average fitness, which provides a much better visual comparison between the three datasets.

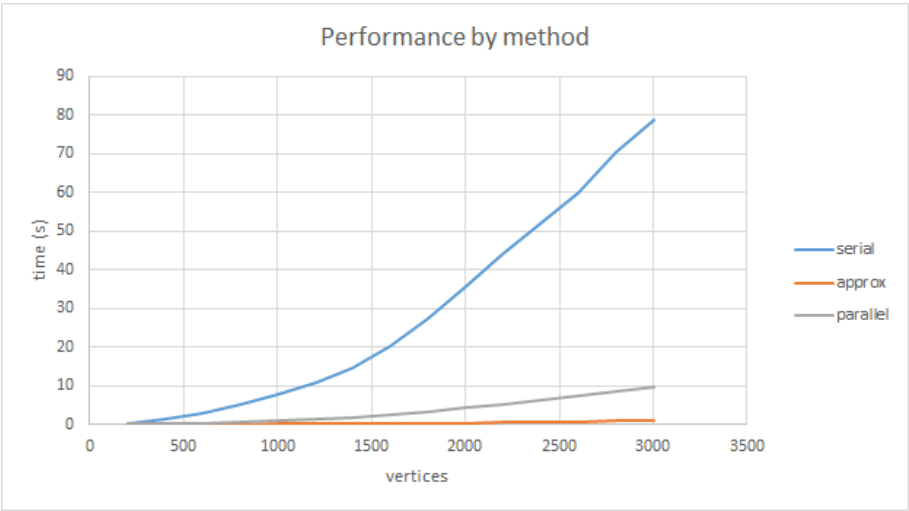

5x10 Torus Relative Last-Gen Fitness (n=100)
fitness - avg
m
serial
approx
parallel


10x10 Torus Relative Last-Gen Fitness (n=100)
fitness - avg
m
serial
approx
parallel



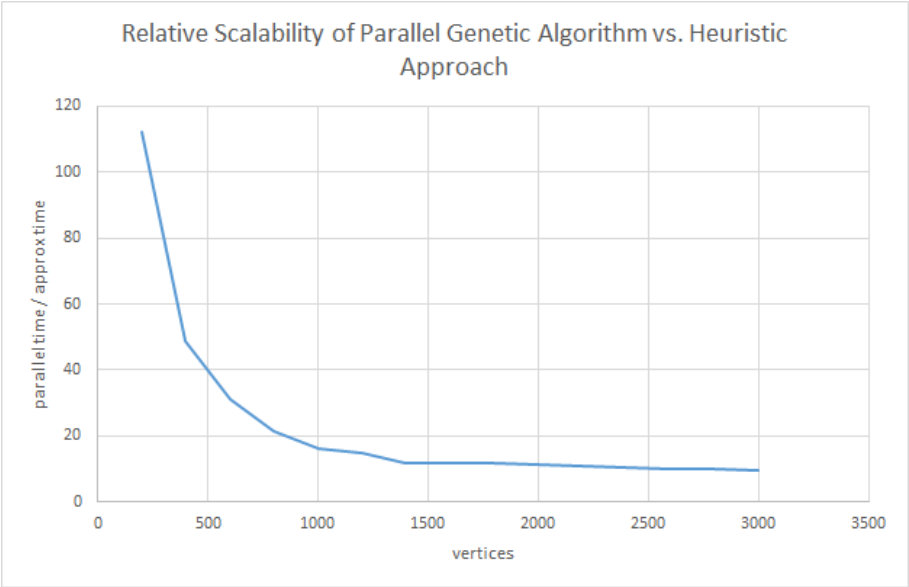
As expected, the genetic algorithm approach yields a better solution than the heuristic approximation for every level of graph-density. This result was anticipated, but tests also reveals that the results are more consistent with a larger torus. This likely results from better isolation of genetic information, which prevents local maxima in the stochastic process from stifling evolutionary advancement.

This graph fails to take into account the computational impact of each approach. In many cases, an inferior solution might be favored over a computationally expensive one. In the next test, we examine the performance and scalability of the different approaches.



Clearly, the heuristic approximation vastly out-performed both the parallelized and serial genetic algorithms in the long run, but further examination reveals that as the number of

vertices increases, the scalability of the parallel method asymptotically approaches that of the heuristic method.



This suggests that for large scale computations, the complexity of the two approaches is roughly equal, making the parallelized genetic algorithm a clear-winner if a high-quality solution is necessary.

4 Conclusion

We already mentioned one approach to improve this design: widen each processor's slice and run several generations of sub-iterations in-between major iterations on the local cells. This helps offset the cost of inter-process communication and waiting for slower processes to catch up between iterations. This offers another opportunity for more parallelization: since each slice's data resides locally on the processor, GPU threading could be used to carry-out the local sub-iterations themselves, further reducing the amount of serial computation.
The biggest challenges of this project involved datatype overflow and MPI inter-process coordination.

5 Submission details

See the file README attached.

6 References

1. *Cantu-Paz, Erick*, A Survey of Parallel Genetic Algorithms

2. *Khuri, Sami et. al*, An Evolutionary Heuristic for the Minimum Vertex Cover Problem