A Parallel Architecture for the Generalized Travelling Salesman Problem: Mid-Year Report

Max Scharrenbroich, maxfs at umd.edu

Dr. Bruce Golden, R. H. Smith School of Business, bgolden at rhsmith.umd.edu

Abstract:
The goal of this project is to develop a parallel implementation of a serial heuristic to attack large instances of the generalized travelling salesman problem (GTSP). By leveraging more computational resources the parallel version of the heuristic is expected to produce higher-quality solutions in less time. A significant portion of this project will involve the development of a parallel architecture that can be extended to host a selected serial heuristic and the GTSP problem class. The extension of the architecture to host the serial heuristic will involve the identification and implementation of different methods of parallel cooperation and levels of parallelism. The parallel heuristic will be tested on a database of problem instances and the performance will be compared to published results of the serial heuristic. In addition, the parallel heuristic will be tested to determine how performance scales with the number of processors used.

1 - Project Background and Introduction

Problem

The generalized traveling salesman problem (GTSP) is a variant of the well-known traveling salesman problem (TSP). Like the TSP, it is a combinatorial optimization problem and has important applications in the field of routing. In the GTSP, a set of nodes or vertices in the plane is grouped into a number of clusters. The goal is to find the shortest tour that visits all the clusters.

More formally, let $G(V,A)$ be a graph where V is the set of vertices and A is the set of arcs. A distance matrix $C = (c_{ij})$ is defined on A. If C is symmetric, the arcs are undirected and can be replaced with edges. In the GTSP, V is partitioned into a set of clusters, $V = \{V_1, V_2, \ldots, V_m\}$, each containing a subset of the nodes from G. The goal is to determine the shortest Hamiltonian tour visiting each cluster exactly once. If the distance matrix is not symmetric, it may be cheaper to visit more than one node in a

cluster. For this project we propose the symmetric version of the GTSP, where V is partitioned into a set of node-disjoint clusters and the distance matrix is symmetric, hence, exactly one node in each cluster is visited. The following figure is an illustration of the problem (Figure 1).

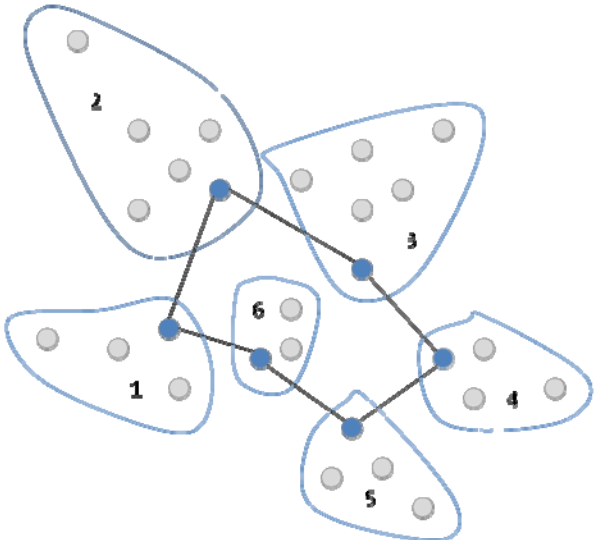

Figure 1 Illustration of the GTSP for a problem with 6 clusters.

Context

Below are real-world examples of GTSP applications:

- Post-box collection and stochastic vehicle routing (G. Laporte, 1996) [5].
- Routing of welfare clients through government agencies (J.P. Saksena, 1970) [8] .
- Warehouse order picking with multiple stock locations (C.E. Noon, 1988) [6].
- Airport selection and routing for courier planes (C.E. Noon, 1988) [6].

Mathematical Formulation

The symmetric GTSP can be formulated as the 0-1 Integer Linear Program (ILP): Given a graph $G(V,E)$, where the set $\{V_1, V_2, \ldots, V_m\}$ is a partition of V into m clusters, and a distance matrix C, where $c_e \in C$ is the Euclidean distance associated with edge $e \in E$ find:

$$\min \sum_{e \in E} c_e x_e$$

Subject to:

$$\sum_{v \in V_k} y_v = 1, \qquad k = 1,2, \ldots m, \qquad (1)$$

$$\sum_{e \in \delta(v)} x_e = 2y_v, \quad \text{for } v \in V, \quad (2)$$

$$\sum_{e \in \delta(S)} x_e \geq 2y_v, \quad \text{for } \emptyset \neq S \subset V, \quad v \in V_s - S, \quad V_s = \{v \in V \mid y_v = 1\}, \quad (3)$$

$$x_e \in \{0,1\}, \quad \text{for } e \in E, \quad (4)$$

$$y_v \in \{0,1\}, \quad \text{for } v \in V, \quad (5)$$

$$x_e = \begin{cases} 1 \text{ if edge } e \text{ is used,} \\ 0 \text{ otherwise, } e \in E \end{cases}$$

$$y_v = \begin{cases} 1 \text{ if vertex } v \text{ is used,} \\ 0 \text{ otherwise, } v \in V \end{cases}$$

$$\delta(S) = \{e = (i,j) \in E \mid i = S, j \in \bar{S}\}$$

Constraint (1) imposes the requirement that each cluster be visited exactly once. The degree equations (2) stipulate that if a vertex v is part of the solution its degree must be equal to two. The subtour elimination constraints (3) ensure the solution does not contain any sub-tours. Constraints (4-5) are the 0-1 integer constraints on the selection of vertices and edges in the solution. $\delta(S)$ is a function defining the edge cut set that partitions the vertex sets S and $\bar{S}$.

Existing Solutions/Algorithms

Like the TSP, the GTSP is NP-hard, and it is conjectured that problems in this class are inherently intractable. Thus, one cannot expect to find "good" or polynomial-time algorithms for solving them. Despite this, there exist exact algorithms for solving the GTSP to optimality. One exact algorithm for solving the GTSP is a branch-and-cut (B&C) algorithm proposed by M. Fischetti in 1997 [4]. Branch-and-cut is a method of combinatorial optimization for solving integer linear programs. The method is a hybrid of branch-and-bound and cutting plane methods.

While B&C techniques drastically reduce the size of the solution space and perform well on small problem instances, these techniques are not polynomial time algorithms. As the size of the problem instance grows, the exponential nature of the problem becomes apparent and B&C algorithms do not terminate in a reasonable amount of time. For example, the run times for the Fischetti B&C algorithm start approaching one day for GTSP problem instances with close to 90 clusters [4].

Heuristic algorithms have been developed to solve larger GTSP problem instances. Heuristic algorithms are search techniques that find approximate solutions to hard

combinatorial optimization problems. The following are three heuristic algorithms that have been successfully applied to the GTSP:

- A Random-Key Genetic Algorithm (L. Snyder and M. Daskin, 2006) [3].
- Generalized Nearest Neighbor Heuristic (C.E. Noon, 1998) [6].
- mrOX Genetic Algorithm (J. Silberholz and B. L. Golden, 2007) [9].

2 - Approach

We propose a parallel approach to assailing the GTSP. Specifically, we will create a parallel architecture and extend the architecture's framework to implement a known and tested serial heuristic algorithm for attacking the GTSP. A new genetic algorithm proposed by J. Silberholz and B.L. Golden in [9], referred to as the mrOX Genetic Algorithm (mrOX GA) [1], has shown promising results and is the chosen heuristic for this project.

In this section an overview of genetic algorithms is given so the reader has some background before giving a description of the mrOX GA. Motivation for parallelizing serial heuristics for combinatorial optimization is outlined, followed by an overview of parallel meta-heuristic classifications. Several methods of parallel cooperation are discussed. A high-level investigation of parallelism in the mrOX GA is given. And finally, the approach for attacking the GTSP and the objectives of the parallel architecture are described.

Overview of Genetic Algorithms

A genetic algorithm is a stochastic search technique commonly used to find approximate solutions to combinatorial optimization problems. Genetic algorithms are a class of evolutionary algorithms that are inspired by the process of natural selection and the theory of evolutionary biology. These algorithms mimic the process of evolution and natural selection by simulating a population of individuals (also known as chromosomes). An iteration of a genetic algorithm is analogous to evolving the next generation of a population. During the iteration a small subset of the fittest individuals (i.e. least cost) are mated to produce offspring with new traits. Since the resulting population is larger than the original, to maintain constant population size a simulated process of natural selection removes individuals that are found to be unfit. This process is iterated through a number of generations until stopping criteria are met.

Initialization:

Initialization is the first step in any genetic algorithm and involves randomly generating many individual solutions to form an initial population. The initial population covers a range of possible solutions (the search space). The population size is typically kept constant from generation to generation and depends on the nature of the problem.

Selection:

A genetic algorithm simulates the evolution of a population from generation to generation and mating of individuals is an important step in this process. Pairs of individuals known as parent chromosomes are selected for breeding from the

population based on fitness and offspring are produced by applying a crossover operator to the pair of chromosomes.

Recombination:

Recombination (crossover) involves the random selection of traits from each parent chromosome for insertion into the child chromosome. A crossover is required to produce viable offspring (feasible solutions for the problem instance). Depending on the structure of the chromosome and the nature of the problem, the crossover by itself is not guaranteed to produce feasible offspring. Thus following the actual crossover, heuristics must be applied to infeasible solutions to ensure that mating always produces feasible offspring.

Local Search:

After recombination there is usually room for additional improvement. It is typical that meta-heuristics perform local search improvement techniques to further improve the offspring. By using local search methods the solutions are guided into the local optimum of the local search neighborhood.

Mutation:

After crossover a small percentage of offspring are selected to be mutated. Mutation involves randomly perturbing parts of an individual's chromosome. As in the case of crossover, mutation must also maintain a solution's feasibility. Mutation ensures diversity in the population and prevents the algorithm from prematurely converging on a poor solution.

Termination:

Due to the combinatorial nature of the problems genetic algorithms are used to solve, there is no convergence analysis that can aid in determining when to terminate the algorithm. There are, however, many types of stopping criteria that can be used for terminating genetic algorithms. A typical stopping criterion is to stop after a fixed number of generations (or after an elapsed time). One method stops the algorithm after the best solution found so far does not change within a fixed number of generations. Another method is to stop after some minimum cost is exceeded.

Overview of the mrOX Genetic Algorithm

The modified rotational ordered crossover genetic algorithm (mrOX GA), proposed by J. Silberholz and B. L. Golden in [9], is a serial genetic algorithm that is specially tailored to the GTSP problem. At its heart is the mrOX crossover operator, which performs a crossover between two parents. In the rest of this section an overview of the mrOX GA is given. For a more detailed treatment of the algorithm and computational results the reader is referred to [9].

It is best to describe the mrOX crossover operator before describing the rest of the mrOX genetic algorithm. First, a description of the ordered crossover (OX) portion of the mrOX is given and then the rotational (r + OX) and modified (m + rOX) portions are discussed so the reader may gain a better understanding of the crossover operator.

Chromosome Representation:

A natural way to represent feasible solutions to the GTSP is with an ordered sequence of nodes (path representation). For example, the sequence {1, 4 , 2} represents the cycle visiting node 1, then node 4, then node 2 and finally back to node 1 to complete the cycle. The path representation lends itself nicely to the idea of a chromosome. Path representations for solutions to the GTSP are also referred to as chromosomes.

OX:

The ordered crossover (OX) operator is based on the TSP ordered crossover proposed by Davis in [3]. The TSP's OX operator randomly selects two cut points on one of two parent chromosomes. The order of the nodes between the two cut points on the first parent is maintained. The remaining non-duplicate nodes from the second parent are placed, in order, starting to the right of the second cut point with wrap-around if necessary. For the GTSP this method is modified so that clusters being added from the second parent do not coincide with clusters from the first parent (i.e. we want to ensure that each cluster is visited only once). Figure 2 shows an illustration of the OX operator as applied to a solution for a hypothetical GTSP.

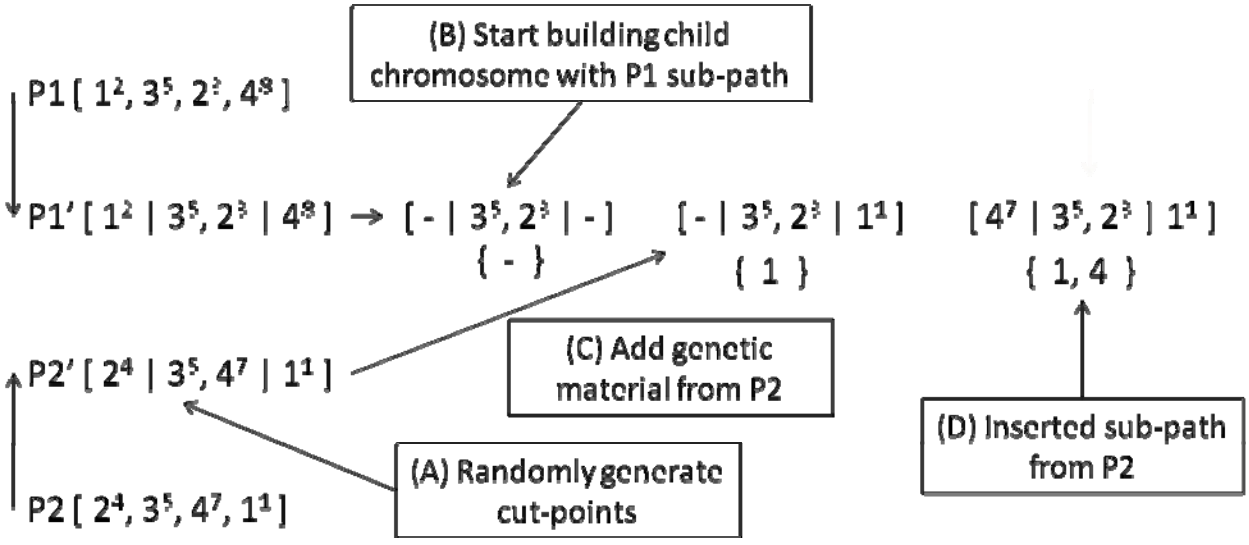

Figure 2 Illustration of OX crossover operator with two parent chromosomes. Each base number represents a cluster with superscripts indicating the number of the node being visited that is part of the cluster. The numbers in the curly brackets represent the ordered list of clusters being added to the new chromosome from the second parent.

In Figure 2 the OX procedure starts with two parent chromosomes, P1 and P2. The square brackets with sequences of numbers represent a chromosome, or solution for a

hypothetical GTSP problem. The numbers represent an ordered pair, (c_i, n_j), where the base number represents a cluster and the superscript indicates the node that is being visited. Initially, cut points are randomly generated on the parent chromosomes (A). In the figure, cut points on the chromosomes are represented by vertical bars and the segmented parent chromosomes are represented by P1' and P2'.

The child chromosome is initialized with the sub-path from the first parent (B). Cluster-node pairs from the second parent, moving left to right, are then added to the empty slots of the child chromosome while avoiding duplicate clusters (C). The curly brackets are a visual aid and show the order in which cluster-node pairs from the second parent are added to the child chromosome. The list of cluster-node pairs from the second parent represents a sub-path to be connected to the first parent's sub-path.

rOX:

Next, the OX is modified with a rotational component yielding the rOX (r + OX). The rotational component acts on the sub-path (from the second parent) to be added to the child chromosome. This sub-path is used to create two sets of sub-paths. One set of sub-paths is generated by applying a shift operator to the original sub-path. The other set of sub-paths is the mirror image of the first set. As an example, assume that after the OX the following sub-path is generated: {1, 2, 3}. Applying a shift operator to this sub-path yields the set of sub-paths:

$$\{1, 2, 3\} \rightarrow \{ \{1, 2, 3\} \{2, 3, 1\} \{3, 1, 2\} \}$$

The second set of sub-paths is the mirror image of the first:

$$\{ \{1, 2, 3\} \{2, 3, 1\} \{3, 1, 2\} \} \rightarrow \{ \{3, 2, 1\} \{1, 3, 2\} \{2, 1, 3\} \}$$

mrOX:

The rotational component is further modified resulting in the mrOX (m + rOX). For each sub-path generated in the rOX, every combination of nodes in the clusters at the end points of the sub-path is generated, resulting in an augmented set of sub-paths to be tested. As an example, suppose one of the sub-paths from the rOX procedure is: $\{ 1^{\{A,B\}}, 3, 2^{\{C,D\}} \}$. Creating the combinations of different nodes at the end points yields the following set of possible sub-paths:

$$\{1, 3, 2\} \rightarrow \{ \{ 1^A, 3, 2^C \} \{ 1^A, 3, 2^D \} \{ 1^B, 3, 2^C \} \{ 1^B, 3, 2^D \} \}$$

An example of the full mrOX crossover is illustrated in Figure 3.

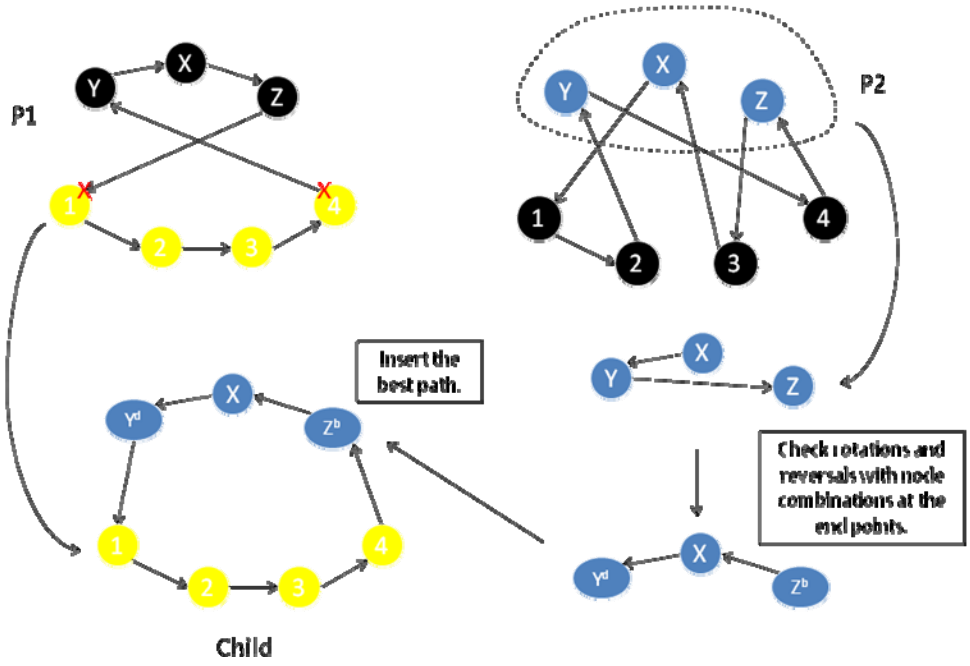

Figure 3 Illustration of mrOX Crossover.

Complexity of the mrOX Crossover:

The complexity of the mrOX crossover operator can be calculated using the following equation:

$$N(S_{p2}) = 2\sum_{i=0}^{m-1} n(i, S_{p2})n\big((i+1) mod(m), S_{p2}\big)$$

Where $N(S_{p2})$ is the number of comparisons required in an mrOX GA crossover operation, $S_{p2} = (c_0, c_1, \ldots, c_{m-1})$ is the ordered set of cluster/node pairs added from the second parent, $m = |S_{p2}|$ is the size of the set, and $n(i, S_{p2})$ is a function that returns the number of nodes in the cluster at index i in the ordered set S_{p2}. The number of comparisons is bounded by:

$$N(S_{p2}) \leq 2mn_{max}^2$$

Where n_{max} is the maximum number of nodes in a cluster. If the number of nodes per cluster is constant throughout all the clusters in the problem, the equation reduces to:

$$N(S_{p2}) = 2mn_c^2$$

Where n_c is the number of nodes in a cluster.

Outline of the mrOX GA:

Having described the mrOX operator, an outline the mrOX GA can now be given.

- Initialization: The mrOX GA starts by initializing seven isolated randomly generated populations (islands) containing 50 individuals each. During the evolution of the isolated populations a light-weight version of the mrOX crossover operator (rOX) followed by local improvement heuristics are applied to quickly generate reasonable solutions. The local improvement involves one full cycle of two-opt followed by one-swap and is applied only to the new best solution in each population.
- Population Merge: After none of the isolated populations produced a new best solution for 10 consecutive generations, the seven isolated populations are merged by selecting the 50 best solutions out of the combined population of 350 solutions.
- Continued Evolution: Post-merge, each generation is evolved using the full mrOX crossover operator followed by local improvement heuristics. The local improvement involves carrying out multiple cycles of two-opt followed by one-swap until no improvements are found. Local improvements are only carried out on child solutions that have better fitness than both parents. Local improvements are also made to a randomly selected 5% of new chromosomes to preserve diversity.
- Reproduction and Death: In each generation a subset 30 individuals are randomly selected using a spinner procedure (based on individual fitness) for reproduction. Each pair of parent chromosomes produces two offspring, yielding a total of 30 child chromosomes. After reproduction, in order to maintain the population size of 50 individuals, 30 individuals are randomly selected for death using a similar procedure to that used for parent selection.
- Mutation: Before and after the merge each chromosome has a 5% probability of being selected for mutation to preserve diversity. The mutation consists of randomly selecting two cut points in the interior of an individual's chromosome and reversing the order of the nodes in between these two points.
- Termination: The algorithm is terminated after the merged population does not produce a better solution for 150 consecutive generations.

Local Search in the mrOX GA:

Local improvement heuristics (also known as local search) are used to find local optima within a neighborhood of a solution and significantly improve the performance of genetic algorithms [10]. In the mrOX GA, local improvement heuristics are applied after the crossover operation. In the initial (light-weight) phase of the mrOX GA, one cycle of 2-opt followed by 1-swap is applied only if the crossover produces a new best solution. In the post-merge phase, full cycles of 2-opt followed by 1-swap are applied only if the crossover produces a solution that is better than both parents. By being selective about applying the local search, the mrOX GA improves run-time by avoiding improvement of solutions that do not appear promising.

The 2-opt improvement heuristic checks every possible two-edge exchange and selects the best one. This is equivalent to uncrossing two crossed paths. The 1-swap inserts a node in every possible position for each of the nodes and picks the best one. Both heuristics have complexity $O(n^2)$. Figure 4 illustrates the two heuristics.

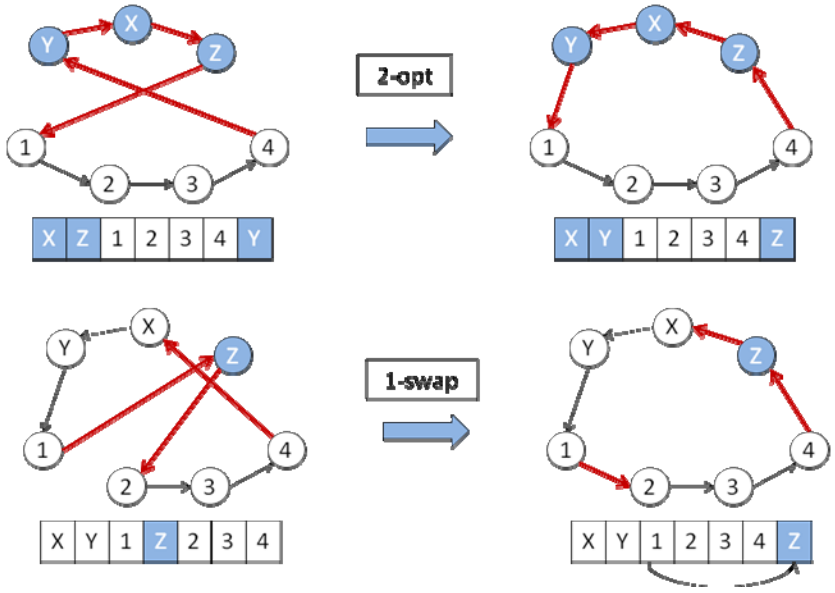

Figure 4 Illustration of 2-opt and 1-swap tour improvement heuristics.

Mutation in the mrOX GA:

As mentioned earlier mutation ensures diversity in the population and prevents the algorithm from prematurely converging on a poor solution. Mutation in the mrOX GA consists of randomly selecting two cut points in the interior of an individual's chromosome and reversing the order of the nodes in between these two points. Figure 5 illustrates a mutation operation.

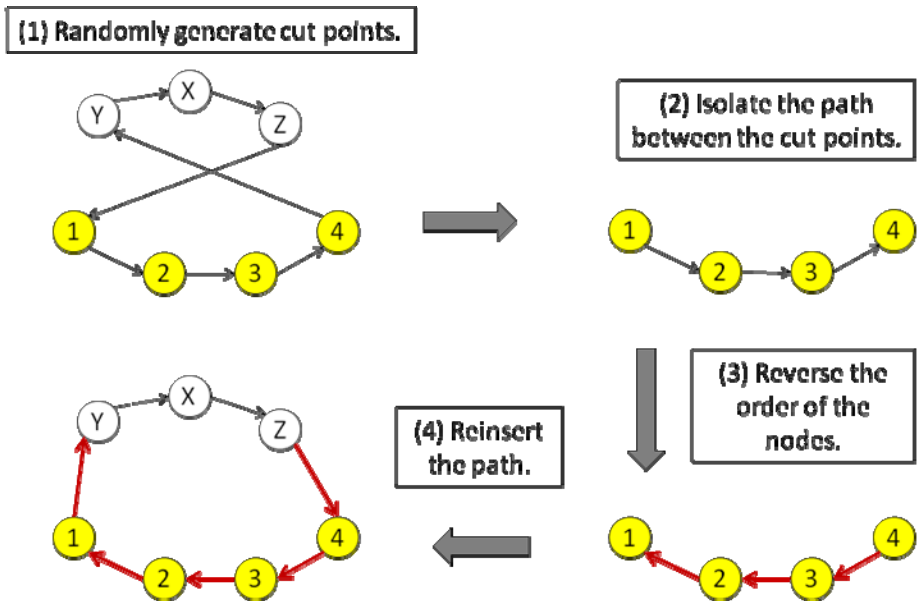

Figure 5 Illustration of mutation in the mrOX GA.

Motivation for Parallelization

Traditionally, the goal when designing parallel algorithms is to reduce the time required to solve the problem. For exact solution methods a useful performance measurement is the speedup, computed as the ratio of the wall-clock time required to solve the problem in parallel with *p* processors and the corresponding solution time taken by the sequential algorithm.

Performance measures such as speedup are harder to define for heuristic methods that are not guaranteed to reach the optimal solution. Thus, the goal of an effective parallel heuristic is to outperform its sequential counterpart in terms of solution quality and computational efficiency [2].

Below are several motivations for parallelizing serial heuristics for combinatorial optimization:

Speedup:

Speedup is an important motivation for parallelizing any algorithm. Simply put, if idle computational resources exist, then they could be put to work producing results faster. An example would be if users needed results in real-time. A parallel implementation may be able to produce results in a matter of seconds instead of minutes or hours.

Increased Problem Size:

Another motivation for parallelization is that by leveraging more computational resources the parallel heuristic can handle larger problem instances.

Robustness with Parameter Exploration:

Many of the meta-heuristics applied to combinatorial optimization have multiple parameters that influence the success of the algorithm on a specific problem or class of problem instances. This can make tuning the parameters to specific problems time consuming, especially if run times are long.

By running different parameterizations on different processes the parameter space can be explored, avoiding the need for manual tuning. In addition, this approach avoids the need for re-tuning when the algorithm is applied to a different problem instance. It is expected that a parallel version of an algorithm using parameter exploration will exhibit robustness and perform consistently on a range of problem instances.

Cooperation:

Parallelization allows cooperation among processes. It is believed that cooperation can improve the solution quality by guiding the search to more promising regions of the search space.

Classification of Parallel Meta-Heuristics

An important step in creating a parallel implementation of a heuristic is in determining what aspects of the heuristic under consideration are amenable to parallelization. In 1998 Crainic and Toulouse proposed three types of classifications for parallel meta-heuristics [1].

- Type-1: Low-Level Parallelism: Attempts to speed up processing within an iteration of a heuristic method. For example, if there is a task within a heuristic that has a high computational burden and can be parallelized then low-level parallelism can be implemented to speed up that portion of the heuristic.
- Type-2: Partitioning of the Solution Space: Partitions the solution space into subsets to explore in parallel. At the end of processing the results are combined in some way to produce the final solution.
- Type-3: Concurrent Exploration: Multiple concurrent explorations of the solution space. Genetic algorithms are particularly amenable to this type of parallelism since these heuristics operate on populations of solutions. In concurrent exploration cooperation among processes can be implemented.

Methods of Cooperation

As mentioned above, hosting a serial heuristic in a parallel architecture allows cooperation to further improve the convergence and quality of a solution. Although there are many ways for cooperation to be implemented, the following three methods of cooperation will be investigated in the course of this project:

No Cooperation:

The case where processes do not use cooperation is a useful benchmark for testing whether or not other methods of cooperation are yielding improvements. In this case there is no exchange of information between the processes. When the stopping criterion is reached the best solution is picked from among the all the processes. Conceptually, this is equivalent to running multiple instances of the serial implementation.

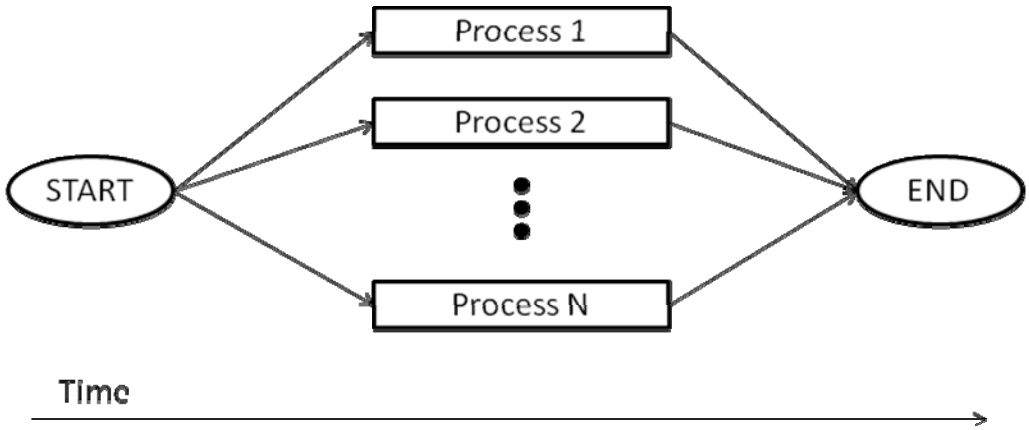

Figure 6 Illustration of no-cooperation scheme.

Solution Warehouse:

The solution warehouse method is a basic architecture for cooperation among worker processes running in parallel. In this method a worker process (solution warehouse) is selected to be the mediator of information between the other worker processes. The solution warehouse collects problem solutions periodically from the worker processes and manages them in a list according to cost (i.e. it keeps track of the best solutions found so far). Due to performance limitations the list is kept to a manageable size. In accordance with a predefined schedule or scheme the solution warehouse sends a subset of the solutions back to the worker processes for further processing. The following is one implementation scheme for the solution warehouse method:

1. Each process sends the best solution to the warehouse after a number of k iterations (or period of time).
2. The warehouse collects the solutions and adds them to a list sorted by the cost, maintaining the top t solutions in memory.
3. The warehouse then assigns the best solution (or subset of solutions) to a subset of the worker processes and then randomly assigns solutions from the list to each remaining processes (with no repeats) for continued processing.

The scheme described above maintains diversity by allowing some of the workers to continue processing solutions that are not necessarily the best found so far. Maintaining diversity prevents premature convergence to poor local optima.

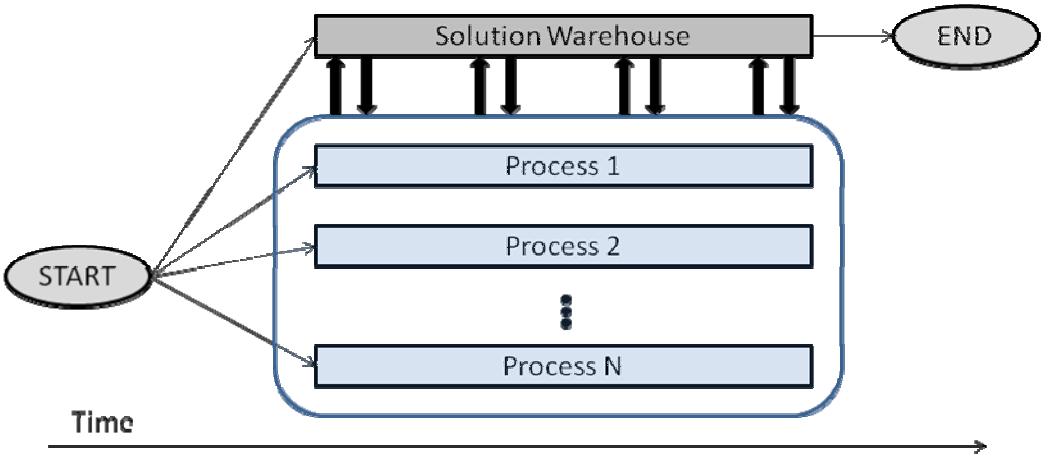

Figure 7 Illustration of solution warehouse method of cooperation. Periodically each process sends a subset of the best solutions it has found so far to the warehouse. The warehouse responds by synchronizing the worker processes to continue working on a subset of the best solutions found by all the processes.

Inter-Worker Cooperation:

Inter-worker cooperation is a general method of cooperation where workers exchange solutions based on a pre-defined communication topology. Workers are only allowed to communicate with their neighbors. An example of a possible communication topology is a unidirectional ring topology. In a unidirectional ring topology each worker sends information to one neighbor. Figure 8 illustrates the ring topology method of communication.

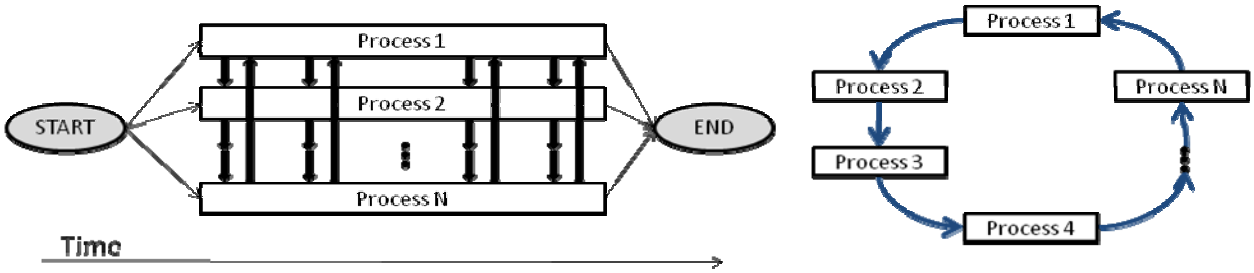

Figure 8 Illustration of inter-worker cooperation in a unidirectional ring topology.

Parallelism in the mrOX GA

The first step in parallelizing an algorithm is in identifying subroutines that are amenable to parallelization. In this section we investigate the ability to exploit different levels of parallelism in the mrOX GA.

Low-Level Parallelism in the mrOX GA:

Recall that low-level parallelism, also known as type-1 parallelism, seeks to speed up processing by parallelizing a computationally intensive subroutine within an iteration of

an algorithm. An ideal candidate for low-level parallelism is a subroutine where the workload can be divided evenly among a number of processors. If the workload can be divided evenly, the computational work will be completed at approximately the same time on all processors, leaving few processors idle as they wait for other processes to complete.

In the mrOX GA there are two computationally intensive subroutines that are candidates for low-level parallelism: the mrOX crossover operation, and the local search improvement heuristics. It was described earlier that the computational complexity of the mrOX crossover operation could be calculated a priori. Therefore, the computational load can be estimated for each crossover operation and the work can be spread across a number of processors in such a way that all processes complete at approximately the same time.

The complexity of the mrOX crossover is a function of the number of cluster/node pairs in S_{pg} and the number of nodes in each of the associated clusters. Unfortunately, the loading for each crossover will be different. This is itself an example of the NP-complete multiprocessor scheduling problem, where given m processors and n jobs, and some time t_i associated with each crossover job j_i, find the schedule that minimizes the amount of idle time. Solving the scheduling problem will take up valuable CPU time, and the resulting schedule will likely leave CPU cycles idle. One idea to overcome the scheduling problem is to allow jobs at the end of the schedule to be terminated early so that all the processes finish at the same time (see Figure 9).

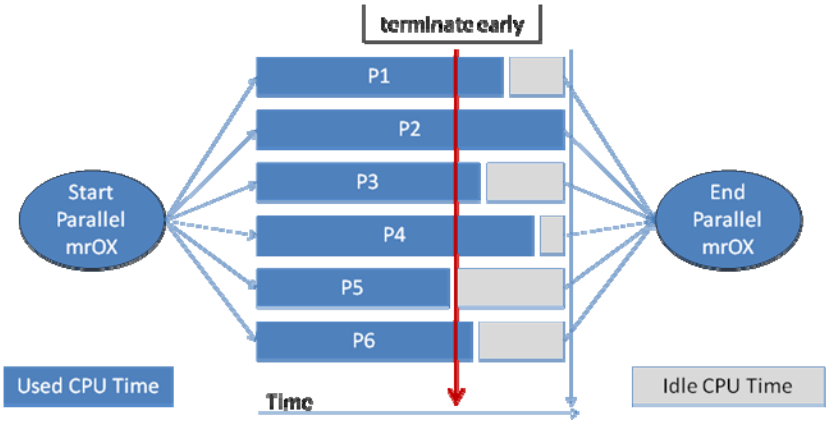

Figure 9 Illustration of load balancing the mrOX crossover over six processors with early termination.

On the other hand, the local improvement heuristic (full cycles of 2-opt followed by 1-swap) is not an ideal candidate for low-level parallelism because the number of cycles of 2-opt and 1-swap are non-determinate. Timing tests of the serial mrOX GA show that a majority of the time is spent in the local search subroutine so even if the crossover operation is parallelized, local search would still be implemented serially (see Figure 10).

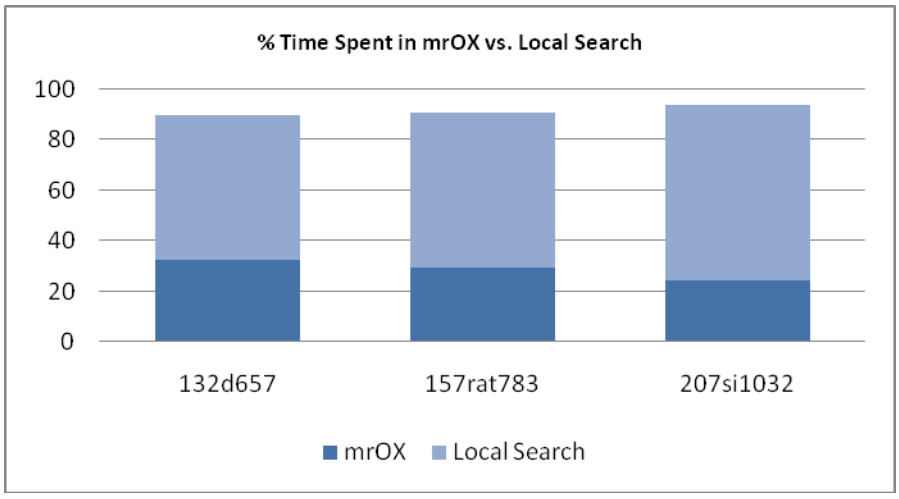

Figure 10 Timing tests for mrOX and local search averaged over five runs of three problems.

While providing a speedup, parallelizing only a subset of computationally intensive subroutines creates a bottleneck and introduces large inefficiencies in a parallel implementation because of idle processors (see Figure 11).

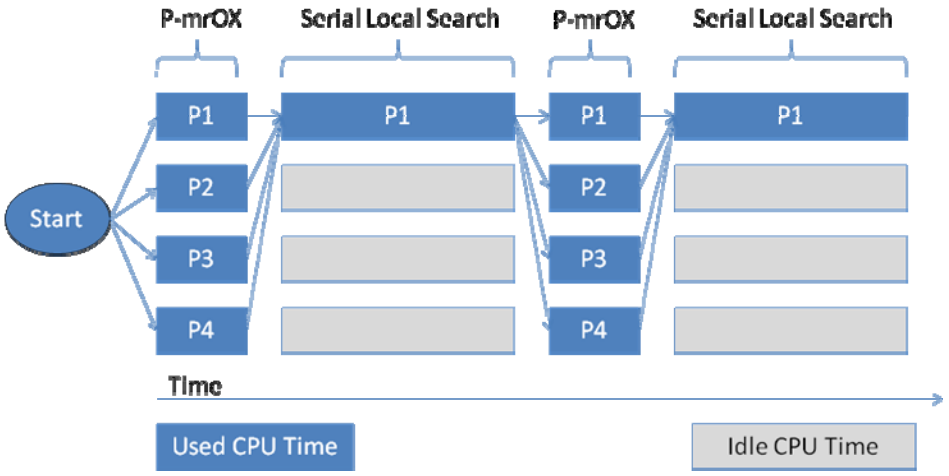

Figure 11 Illustration of inefficiencies with parallel mrOX (P-mrOX) and serial local search.

This investigation into low-level parallelism in the mrOX GA suggests that it is not amenable to this kind of parallelism. The main avenue for parallelization in the mrOX GA will be through concurrent exploration.

Concurrent Exploration:

Type-3 parallelization amounts to multiple concurrent explorations of the solution space. This is equivalent to running multiple copies of a serial heuristic in parallel. Genetic algorithms are particularly amenable to this type of parallelism since these heuristics operate on populations of solutions. In a type-3 parallel implementation cooperation

among processes can be implemented. A study of fine-grained or cellular genetic algorithms (cGAs) was undertaken as motivation for parallel cooperation schemes.

Cellular genetic algorithms are genetic algorithms where the population is structured on an N-dimensional toroidal mesh (N is typically 2). Each individual can only reproduce with other members of its neighborhood (e.g. up, down, left and right – see Figure 12). In this type of population structure neighborhoods overlap, giving cGAs an implicit mechanism of migration. Thus, more successful solutions diffuse more rapidly through the population. Diffusion of solutions in cGAs is much slower than in panmictic GAs, and it is reasoned that because of slower diffusion, cGAs foster the formation of niches. The formation of niches preserves diversity (exploration) in the overall population while promoting specialization (exploitation) within these areas [11].

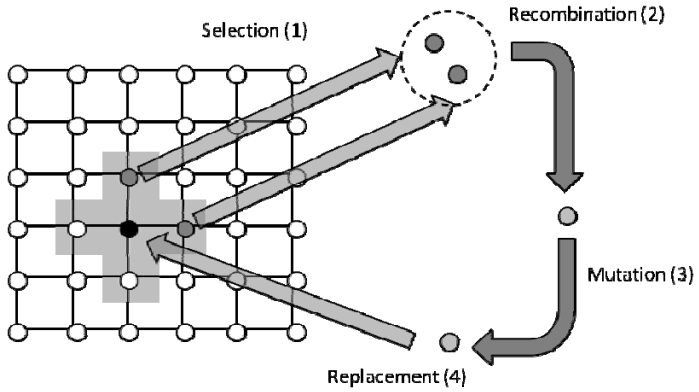

Figure 12 Illustration of reproduction in a cGA on a 2-D toroidal mesh.

Implementing the mrOX GA using the population structure of a cGA would significantly alter it so as to make it a different serial algorithm. Applying the mesh communication topology at a process level could be successful, and would allow solutions to migrate (diffuse) between neighboring processes, providing a similar mechanism of niche formation as in cGAs (Figure 13). The mesh communication topology will be further considered for the parallel mrOX implementation.

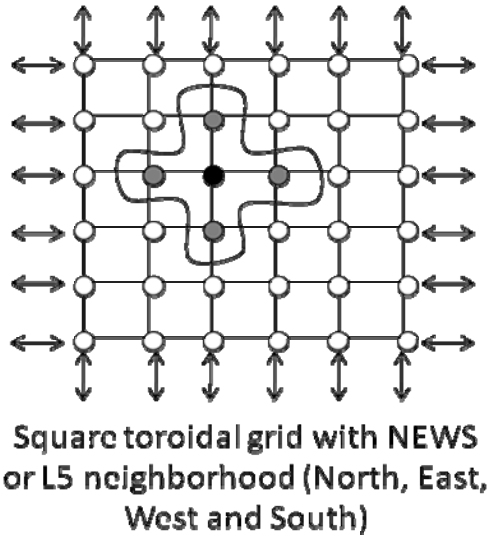

Figure 13 Illustration of a NEWS neighborhood of processors on a 2-D toroidal grid.

Some further investigation and testing will be required to determine a reliable communication scheme over the mesh topology. Some important parameters that will be considered will be communication interval as well as parameters influencing the selection of neighbor solutions.

Method of Approach

The following list outlines the method of approach we will take for creating a parallel heuristic for the GTSP:

1. Develop a general parallel architecture for hosting sequential heuristic algorithms.
2. Extend the framework provided by the architecture to host the mrOX GA heuristic and the GTSP problem class.
3. Since genetic algorithms are well suited for the type 3 parallelization (concurrent exploration) the parallel implementation will consist of concurrent processes running the mrOX GA.
4. Type 1 or low-level parallelism will be considered in addition to the type 3 parallelism mentioned above.
5. Implement several different methods of parallel cooperation.

Parallel Architecture Objectives

The following is a list of objectives of the proposed parallel architecture:

- Provide a layer of abstraction from Message Passing Interface (MPI) so application developers do not need to be aware of the MPI implementation details.
- Provide a framework of interfaces, classes and event handlers for extensibility.

- Provide parallel cooperation using the selected cooperation scheme.
- Utilize multi-threading for handling I/O and framework related tasks on idle CPUs to prevent processing interruptions.
- Provide a capability for reporting process resource usage, status, debug and timing information.

3 – Implementation

Hardware and Software:

Initial Development and Testing: Single processor PC running Linux O/S, then move to multi-core PC.

Final Testing: UMD's Deepthought Cluster, Linux O/S, with up to 64 nodes where each node has at least 2 processors.

Language and Libraries: C/C++, POSIX Threads and MPI Libraries.

Preliminary Design:

The following is a preliminary design for the parallel architecture and framework.

Summary of Architecture:

The architecture is made up of three levels:

1. MPI/communications layer
2. Framework layer
3. Extension of framework

MPI/Communications Layer:

Message – a base class for passing objects and data structures to/from processes. This class is like a serialization interface that creates messages to/from a byte stream.

Process – a base class for all other process types.

RootProcess – a root process manages a group of worker processes. The root process is responsible for collecting messages and information produced by worker processes.

WorkerProcess – a process that takes initialization parameters, periodic input, produces periodic output and a final output. Worker processes expose virtual functions for handling initialization, periodic input, and periodic output. An important function is the WorkerThread function that is executed after initialization. This function is defined by child classes.

WorkGroup – a class that encapsulates a RootProcess and a number of WorkerProcesses. This class has a static factory function for creating WorkGroup instances.

InterWorkerTopology – a class that manages the inter-worker communication topology for WorkerProcesses within a WorkGroup. An example is the Cartesian MPI communicator pattern.

Neighborhood – a class that defines the communication neighborhood of a specific WorkerProcess.

Framework Layer:

ProblemInstance – a base class for encapsulating specific problem instances. Functions to set whether the problem is a minimization or maximization problem type.

SerialHeuristic – a base class for encapsulating a specific serial heuristic. A serial heuristic has a collection of Phases – or execution states. Any preexisting serial heuristic must be broken down into at least one Phase. A Phase consists of an execution of a single iteration of the specific phase of a serial heuristic.

Solution – a base class for encapsulating an individual solution. This class exposes a function for computing or reporting the solution's objective function value.

SolutionSet – a collection of solutions (population of solutions). This provides functions for sorting and collecting statistics on the solutions within the set. Functions like BestSolution(), WorstSolution(), MaxObjectiveValue(), MinObjectiveValue(), MeanObjectiveValue(), MedianObjectiveValue(), and StddevObjectiveValues(). In addition, the class provides the ability to select solutions randomly using several methods.

Extension of Framework:

ProblemInstance, Message::GTSP – Generalized traveling salesman problem instance.

Solution, Message::GTSPSolution – An instance of a solution to the GTSP.

SolutionSet<GTSPSolution> - A collection of GTSPSolutions.

WorkerProcess [SerialHeuristic::mrOXGA] – The mrOX genetic algorithm heuristic.

Phase1 – Population isolation phase.

Phase2 – Post-merge population evolution.

Cooperation1 – Exchange solutions with neighbors.

Cooperation2 – If the current worker process stagnates and other workers not part of the current neighborhood are making improvements this cooperation phase makes a drastic population shift to incorporate solutions from more successful processes.

4 - Databases

The database for testing the parallel algorithm will be based on a subset of TSP instances from the well-known TSPLib[1], a library of TSP instances that can be found online. We shall use an existing code that implements the method described in Section 6 of [4] to cluster nodes of a TSP instance. This method clusters nodes based on proximity to each other, iteratively selecting $m = \lceil n/5 \rceil$ centers of clusters such that each center maximizes its distance from the closest already-selected center. Then, all n nodes are added to the cluster whose center is closest.

Fischetti et al.'s branch-and-cut algorithm provides exact values for TSPLib datasets where the number of nodes ranged between 48 and 400 nodes and the number of clusters is between 10 and 80 respectively [4]. The serial heuristic run times for these problem instances are fairly short (all less than 10 seconds) and we don't expect the parallel implementation to perform better than the serial one due to lack of search depth and parallelization overhead. In [9] the mrOX GA was tested against another genetic algorithm, Snyder and Daskin's Random-Key Genetic Algorithm [10], on problem instances where the number of nodes is between 400 and 1084 and the number of clusters is between 80 and 200 respectively. In this set of instances the run time for the serial algorithm ranged from 10 to 131 seconds. It is for this set of instances that we will test performance and where we expect to see improvement using the parallel implementation.

5 – Validation and Testing

Validation and testing will consist of several phases.

Validation

Validation is important step in verifying that the behavior of the software code matches what it is intended to do. The following procedure will be used to validate the code.

1. Validate the parallel architecture using a simple test algorithm and generate several test-cases to test the functionality of the parallel architecture.

[1] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

2. Test the parallel implementation using one processor over a number of runs for a subset of problem instances and compare those results to published ones. It is expected that run times and results should match closely to the published ones.
3. Test the parallel implementation with more than one processor over a number of runs for the same subset of problem instances used in part 2.

Testing

After validation we will test the performance of the parallel implementation to the serial one. As mentioned earlier, comparing a parallel heuristic to its serial counterpart is not so straight forward. We propose the following set of tests to measure performance improvements due to parallelization. For the parallel implementation and the selected cooperation scheme run the following tests:

1. For one set of tests use the published results of solution costs for runs of the serial algorithm in [9] as a stopping criterion for the parallel implementation.
2. Run the parallel implementation with different numbers of processors and measure the processing times using the above stopping criteria.
3. Compare the processing times to the ideal processing time as a function of the number of processors. The ideal processing time is computed as the ratio of the serial processing time and the number of processors.
4. For testing the efficacy of cooperation run the above tests using a parallel implementation and the non-cooperative scheme. Compare the results to the cooperative scheme. Conceptually, this is equivalent to the serial implementation.
5. Reduce the population size and/or the number of parents selected for crossover and measure the processing time and solution quality.

6 - Project Schedule/Milestones

October 16-30: Start design of the parallel architecture.

November: Finish design and start coding and testing of the parallel architecture.

December and January: Continue coding parallel architecture and extend the framework for the mrOX GA algorithm and the GTSP problem class.

February 1-15: Begin test and validation on multi-core PC.

February 16-March 1: Move testing to Deepthought cluster.

March: Perform final testing on full data sets and collect results.

April-May: Generate parallel architecture API documentation, write final report.

7 – Deliverables

- Parallel architecture code, scripts and API documentation.
- Tables of results.
- Final report.

8 – References

1. Crainic, T.G. and Toulouse, M. Parallel Strategies for Meta-Heuristics. *Fleet Management and Logistics*, 205-251, 1998.
2. Crainic, T.G. Parallel Solution Methods for Vehicle Routing Problems. *Operatioins Research* 43, 171-198, 2008.
3. L. Davis. Applying Adaptive Algorithms to Epistatic Domains. *Proceeding of the International Joint Conference on Artificial Intelligence*, 162-164, 1985.
4. M. Fischetti, J.J. Salazar-Gonzalez, P. Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research* 45 (3): 378–394, 1997.
5. G. Laporte, A. Asef-Vaziri, C. Sriskandarajah. Some Applications of the Generalized Traveling Salesman Problem. *Journal of the Operational Research Society* 47: 1461-1467, 1996.
6. C.E. Noon. The generalized traveling salesman problem. Ph. D. Dissertation, University of Michigan, 1988.
7. C.E. Noon. A Lagrangian based approach for the asymmetric generalized traveling salesman problem. *Operations Research* 39 (4): 623-632, 1990.
8. J.P. Saksena. Mathematical model of scheduling clients through welfare agencies. *CORS Journal* 8: 185-200, 1970.
9. J. Silberholz and B.L. Golden. The Generalized Traveling Salesman Problem: A New Genetic Algorithm Approach. *Operations Research/Computer Science Interfaces Series* 37: 165-181, 2007.
10. L. Snyder and M. Daskin. *A* random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational Research* 17 (1): 38-53, 2006.
11. E. Alba and B. Dorronsoro. Cellular Genetic Algorithms, *Operations Research/Computer Science Interfaces*, 2008, Springer Science and Business Media, LLC.