Implementation and Parallelization of an Evolutionary Algorithm for Constrained Optimization Problems

GRADUATE PROJECT TECHNICAL REPORT

Submitted to the Faculty of
the Department of Computing Sciences
Texas A&M University-Corpus Christi
Corpus Christi, TX

in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

by

Roja Ramani Molupuju
Spring 2011

Committee Members

Dr. Michael C. Scherger
Committee Chairperson

Dr. P. A. Simionescu
Committee Member

Dr. David Thomas
Committee Member
ABSTRACT

The general nonlinear programming (NLP) problem requires finding the optimum point (minimum or maximum) of a function of $n$ real variables subjected to some given constraints. There are numerous situations in science and engineering where the optimum is bounded adding complexity to the optimization problem. Solving the general NLP problem analytically is possible only in a limited number of cases, so numerical methods are therefore used. Of these methods, evolutionary algorithms are very promising, because of their robustness, ability to deal with multi-modal and noisy functions and being very well suited to parallelization.

Solving constrained nonlinear programming problems using evolutionary computations is considered. An approach for this is quoted in the literature [Simionescu 2004]. According to the method two populations are evolved, one population (females) is evolved inside the feasible domain of the design space and a second population (males) is evolved outside this feasible domain. Both populations can be independently subject to crossover and mutation operations and the design space explored. Female-male crossover however ensures the desirable increase in the search pressure upon the boundaries of the feasible space - it is known that in many optimization problems the global optimum is bounded. This will be explored using some test functions.

A parallelized version of this algorithm is implemented part of this project, and its effectiveness is tested using benchmark problems taken from literature.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>01</td>
</tr>
<tr>
<td>2. Background And Related Research</td>
<td>04</td>
</tr>
<tr>
<td>2.1 Evolutionary Computation</td>
<td>05</td>
</tr>
<tr>
<td>2.2 Varieties of Evolutionary Algorithms</td>
<td>08</td>
</tr>
<tr>
<td>2.2.1 Evolutionary Programming</td>
<td>08</td>
</tr>
<tr>
<td>2.2.2 Evolution Strategies</td>
<td>08</td>
</tr>
<tr>
<td>2.2.3 Genetic Algorithm</td>
<td>08</td>
</tr>
<tr>
<td>2.3 Message Passing Interface (MPI)</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1 Reasons For Using MPI</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 Programming Model</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Basic Features of Message Passing Programs</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1 A Generic MPI Program</td>
<td>14</td>
</tr>
<tr>
<td>2.4.2 Communicators</td>
<td>15</td>
</tr>
<tr>
<td>2.4.3 Basic MPI Routines</td>
<td>16</td>
</tr>
<tr>
<td>3. Narrative</td>
<td>17</td>
</tr>
<tr>
<td>4. Project Design</td>
<td>18</td>
</tr>
</tbody>
</table>
4.1 Two Population Evolutionary Algorithm ........................................ 18
4.2 Parallelization ............................................................................. 23
  4.2.1 Parallelization using Manager-Worker Paradigm ..................... 23
5  Test Functions ................................................................................ 25
  5.1 Rosenbrock Function .................................................................... 25
  5.2 Benacer-Tao Function .................................................................... 31
  5.3 Rosen-Suzuki Fist Function .......................................................... 36
  5.4 Himmelblau Nonlinear Optimization Problem ............................... 40
6  Summary of Results ......................................................................... 42
7  Conclusion ....................................................................................... 44
8  Future Work ..................................................................................... 46
Acknowledgements ............................................................................... 47
References ......................................................................................... 48
Appendices ......................................................................................... 53
  1. main.cpp ....................................................................................... 53
  2. functions.h .................................................................................... 65
LIST OF FIGURES

1. Manager-Worker Paradigm ............................................................... 24
2. Plot of Rosenbrock Function (in two variables) with constraints ................. 26
3. Number of processes vs. Time for Algorithm for Rosenbrock’s function ........ 27
4. Number of processes vs. Total Time for Rosenbrock’s function ................... 28
5. Single graph showing all the computation times for different number of processes for Rosenbrock’s function ............................................................... 29
6. Graph showing optimum point for Rosenbrock’s function ............................ 30
7. Plot of Benacer-Tao (in two variables) with constraints ................................. 31
8. Number of processes vs. Time for Algorithm for Benacer-Tao function .......... 33
9. Number of Processes vs. Total Time graph for Benacer-Tao function .............. 34
10. Single graph showing all the computation times for different number of processes for Benacer-Tao function ............................................................ 34
11. Graph showing the optimum point for Benacer-Tao function ........................ 35
12. Number of processes vs. Time for Algorithm for Rosen-Suzuki Fist function ............................................................... 37
13. Number of processes vs. Total time graph for Rosen-Suzuki Fist function .......... 38
14. Single graph showing all the computation times for different number of processes for Rosen-Suzuki function ............................................................ 39
LIST OF TABLES

1. Table showing the basic MPI routines ................................................................. 16

2. Table showing values for Rosenbrock’s Function for different
   number of processes .......................................................................................... 26

3. Table showing the values for BenacerTao function for different
   number of processes .......................................................................................... 32

4. Table showing the values for RosenSuzuki function for different
   number of processes .......................................................................................... 37

5. Comparison of current algorithm with previous techniques
   for Himmelblau Problem ..................................................................................... 41

6. Evaluation of functions for different nF, nM, nG values ..................................... 42
1. INTRODUCTION

Many search and optimization problems in science and engineering involve a number of constraints which the optimal solution must satisfy. A constrained optimization problem is usually written as a nonlinear programming (NLP) problem of the following type:

Minimize \( F(x_1, \ldots, x_i, \ldots, x_n) \) \( (1 \leq i \leq n) \)  

subject to the following side constraints:

\( x_{i_{\min}} \leq x_i \leq x_{i_{\max}} \) \( (1 \leq i \leq n) \)  

and inequality and equality constraints:

\( g_j(x_1, \ldots, x_n) \leq 0 \) \( (1 \leq j \leq m) \)  
\( h_k(x_1, \ldots, x_n) = 0 \) \( (1 \leq j \leq p) \).

In the above NLP problem, there are \( n \) variables (that is, \( X \) is a vector of size \( n \)), \( m \) inequality constraints, and \( p \) equality constraints. The function \( F(X) \) is the objective function, \( g_j(X) \) is the \( j \)th inequality constraint, and \( h_k(X) \) is the \( k \)th equality constraint. The \( i \)th variable varies in the range \( x_{i_{\min}} : x_{i_{\max}} \).

Equality constraints could be eliminated with some of the variables [Back 2000] [Papalambros 2000], only NLP problems that are subject to inequality constraints are considered in this project.

There are many evolutionary algorithms that were introduced by researchers by which the general NLP problem could be solved. One of the main differences between various approaches lay in constraint-handling the techniques employed as follows: 1) various implementations of the penalty function method, 2) specialized representations and operators, 3) repair algorithms, 4) separation of objectives and
constraints (behavioral memory, superiority of feasible points, multi-objective optimization techniques) 5) hybrid algorithms etc.

Precisely searching the boundary area between the feasible and the infeasible regions is most critical for any function to find the global optimum. The inability of evolutionary systems to search precisely the boundary area is the main reason for difficulty in locating the global optimum. Some of the constraints are active at the global optimum. For this reason, in may constrained optimization problems, it is more difficult to locate the global optimum [Michalewicz 1996]. A two-population evolutionary computation approach was proposed in paper [Simionescu 2004] in which two population of individuals are evolved – one of feasible and the other of infeasible by which the search pressure upon the boundaries of the feasible space can be increased. Feasible individuals (also called females) are the ones that are evolved inside (including the boundaries of the feasible space), while the infeasible individuals (also called males) are evolved outside the feasible space. These two populations are subject to mutation and asexual crossover. Female-male crossover ensures the search pressure on the boundaries of the feasible space. This crossover can be performed between two or more feasible and infeasible individuals, and with, or without, favoring the better fit females or the better ranked males.

When two populations, one of feasible and one of infeasible individuals, are distinctively evolved, and these two populations interact systematically (rather than occasionally) for the purpose of exploring the boundaries of the feasible space. Since the criterion based on which individuals are assigned to the two populations does not change during evolution, the behavior of the two populations is easier to understand.

The two-population evolutionary algorithm is described in the section 5, it was proposed by [Simionescu 2004]. This algorithm was initially tested for functions with
two variables; this project works with the algorithm for functions with multiple
variables and is tested. This algorithm is implemented using C++ programming using
Message Passing Interface (MPI). This is a parallel implementation of the algorithm.
It is run for several benchmark problems from literature. Evaluation of the several
test objective functions with multiple variables and the comparisons with the values
from the literature are also provided.
2. BACKGROUND AND RELATED RESEARCH

Evolutionary algorithms have been applied as a choice of solving various optimization problems using evolution strategies with some promising results [Eiben 2002]. These algorithms have features like solving problem without having a complete knowledge on the problem being solved, implementation of a problem is very easy, and finally they are inherently parallel. Evolutionary algorithms have many advantages over conventional nonlinear programming techniques: the gradients of the cost function and constraints functions are not required, and the chance of being trapped by a local minimum is lower [Michalewicz 1984]. Nonetheless, the algorithms that are currently in existence are not capable to handle constrained optimization problems. Specifically, their accuracy in terms of numerical values is low when compared to deterministic nonlinear programming techniques.

Michalewicz and Schoenauer [1996] have presented a comparison study of existing evolutionary algorithms developed for solving constrained optimization problems. According to this study, the existing algorithms are grouped as follows: 1) methods based on preserving feasibility of solutions, 2) methods based on penalty functions, 3) methods based on the superiority of feasible solutions over infeasible solutions, and 4) other hybrid methods. As in the deterministic nonlinear programming techniques, handling various types of constraints is the key task to be taken in to consideration. As shown in [Michalewicz 1996] [Papalambros 1983], the disadvantage with these constraints are they doesn’t scale well for all set of problems i.e., if they work good for one set of problems then they don’t work for other. Note that all of the existing evolutionary algorithms are based on evolution of a single population. One exception is the work of [Paredis 1997], which suggests a solution for constraint satisfaction i.e., co-evolution. Although this work does not directly
address constrained optimization problems, it is the first attempt known to use the concept of co-evolution to handle constraints.

2.1 Evolutionary Computation

Evolutionary computation techniques have attained a great importance in dealing with complex numerical functions using optimization techniques. However, these techniques did not gain any importance in the field of non-linear programming because of fact they have not addressed the issue of constraints in a systematic way. Only recently have several methods been proposed for handling nonlinear constraints by evolutionary algorithms for numerical optimization problems [Grefenstette 1985].

Evolutionary computation is a process of repetitive progress, such as growth or development in a population. This population is then selected in a guided random search using parallel processing to achieve the desired end. A larger number of variety evolutionary computational models are in existence, which is referred to as evolutionary algorithms [Chang 2009]. All models have some commonalities’ in sharing the conceptual base of simulating the evolution using the processes of selection and reproduction. These processes depend on the perceived performance (fitness) of the individual structures as defined by an environment.

Early analogies between the mechanism of natural selection and a learning (or optimization) process led to the development of the so-called “evolutionary algorithms” (EAs) [Papalambros 2000], which simulate the evolutionary process in a computer. Evolutionary algorithms typically have three main streams whose motivations and the way they got in to existence are different from each other: evolution strategies [Eiben 2002], evolutionary programming [Carlos 1996], and genetic algorithms [Gen 1997]. Now a days, the change has been taken place to
decrease the differences among these three streams and refer all of them in to a simple form called evolutionary algorithms when talking about any of them.

In evolutionary algorithm the population is randomly initialized, although domain specific knowledge can also be used to bias the search. The fitness of an individual is measured using evaluation according to its importance in the search space. Evaluation may be in two ways calculating fitness function, which is very simple task and the other one is elaborate simulation where it seems to be complex task [Jong 1993]. After initializing the population randomly the next set of task is to make selection, where it is done in two steps parent selection and survival. Parent selection decides who become parents and how many children the parents have. Children are created via recombination, which exchanges information between parents, and mutation, which further perturbs the children. The children are then evaluated. Finally, the survival step decides who survives in the population.

In general, we need the following basic components to implement an EA in order to solve a problem [Jong1 1993]:

1. A representation of the potential solutions to the problem.
2. A way to create an initial population of potential solutions (this is normally done randomly, but deterministic approaches can also be used).
3. An evaluation function that plays the role of the environment, rating solutions in terms of their “fitness”.
4. A selection procedure that chooses the parents that will reproduce.
5. Evolutionary operators that alter the composition of offspring (normally, crossover and mutation).
6. Values for various parameters that the evolutionary algorithm uses (population size, probabilities of applying evolutionary operators, etc.).
Evolutionary algorithms are advantageous in many applications. However, these advantages are generally not taken into consideration when the algorithms are unconstrained optimization procedures, this gives us an conclusion of incorporating constraints into fitness function.

One of the frequently used common methods of incorporating constraints is by having penalty functions in EA. However, they have some difficulties where these difficulties are overcome by using good penalty factors, they are proposed by researchers in evolutionary computing. Besides this, researchers also developed different ways to handle constraints. Mainly the constraints are developed to handle most complex optimization problems in which it is highly impossible to estimate good penalty factors or to generate a single feasible solution.

There are several surveys on constraint-handling techniques available in the specialized literature, but they are either too narrow (i.e., they cover a single group of constraint-handling techniques) or they focus more on empirical comparisons and on the design of interesting test functions [Back 2000] [Carlos 1996]. None of these surveys attempt to focus on the discussion of the different aspects of each method.
2.2 Varieties of Evolutionary Algorithms

The origins of evolutionary algorithms can be traced to at least the 1950’s (e.g., Fraser, 1957; Box, 1957). For the sake of brevity, main methodologies that have emerged in the last few decades: "evolutionary programming" (Fogel et al., 1966), "evolution strategies" (Rechenberg, 1973), and "genetic algorithms" (Holland, 1975) are mentioned here.

2.2.1 Evolutionary Programming

Evolutionary programming (EP), developed by Fogel et al. (1966) traditionally has used representations that are tailored to the problem domain. For example, in real-valued optimization problems, the individuals within the population are real-valued vectors [Jong 1993]. Similarly, ordered lists are used for traveling salesman problems, and graphs for applications with finite state machines. EP is often used as an optimizer, although it arose from the desire to generate machine intelligence.

2.2.2 Evolution Strategies

Evolution strategies (ESs) were independently developed by Rechenberg (1973), with selection, mutation, and a population of size one. Schwefel (1981) introduced recombination and populations with more than one individual [Jong 1992], and provided a nice comparison of ESs with more traditional optimization techniques. Due to initial interest in hydrodynamic optimization problems, evolution strategies typically use real-valued vector representations.

2.2.3 Genetic Algorithms

Genetic algorithms (GAs), developed by Holland (1975), have traditionally used a more domain independent representation, namely, bit-strings. However, many
recent applications of GAs have focused on other representations, such as graphs (neural networks), Lisp expressions, ordered lists, and real-valued vectors.

In general, GAs are highly simplified and abstract computational models inspired by natural selection [Gen 1997]. The differences in "fitness calculation" in GAs and nature clearly illustrate this. Standard GAs apply an a priori defined fitness function (e.g. the function one wants to optimize) to an individual. They typically use an "all at once" calculation: individuals are evaluated immediately after their creation (i.e. birth). Fitness calculation in nature is substantially different [Adewuya 1993]. It consists of a continuous series of tests during an individual's life. Furthermore, these tests are not strictly pre-defined. They originate from a complex environment. This environment is not only influenced by the animal's own actions but also by the other individuals as well as other processes occurring in the world (e.g. climatologically or geophysical changes). Summarizing one can say that - in contrast with GAs - nature uses a far more partial but continuous fitness evaluation in order to adapt to a complex world.

In nature, various feedback mechanisms between the individuals undergoing selection provide a strong driving force towards complexity [Eiben 2002]. Predator-prey relations are the most well-known examples. There is a strong evolutionary pressure for prey to defend themselves better (e.g. by running quicker, growing bigger shields, better camouflage ...) [Carlos 1996] [Back 2000]. In such arms races, success on one side is felt by the other side as failure to which must be "responded" in order to maintain one's chances of survival. This, in turn, calls for a reaction of the other side. This process of co-evolution can result in a stepwise increase in complexity of both predator and prey. Hillis was the first to propose the computational use of predator-prey co-evolution. Here, we will see that the
introduction of a partial, continuous fitness evaluation results in a robust, fine-grained, algorithm with a significantly increased performance. It also gives rise to a general co-evolutionary framework able to address traditional problems in Computer Science and Artificial Intelligence. This new type of GA is called a *Co-evolutionary Genetic Algorithm* (CGA). Two earlier applications based on predator-prey interactions illustrate the approach: the search for good classification neural networks and the search for solutions to constraint satisfaction problems. Earlier papers [Carlos 1999] [Papalambros 1983] describe these applications in greater detail and compare the performance of a CGA with other GAs. Now, these applications are only used to illustrate the abstract CGA introduced here. Both examples use predator-prey relations to improve the power of artificial search. Obviously, many other mechanisms - not necessarily based on an inverse fitness interaction between members of the different species - exist in nature [Gen 1997]. Symbiosis is such an important and widely occurring counterexample. It consists of a positive fitness feedback in which a success on one side improves the chances of survival of the other.

One of the key problems for using GAs in practical applications is to design the fitness function, particularly when we do not know where the global optimum is located [Jong 1993] [Kim 1997] [Tahk 2000]. A comparative estimate of how good as a solution turns out to be enough in most case (e.g. the largest value has to be closer to the global maximum if we are trying to maximize the objective function), but if we are dealing with constrained problems, we have to find a way of estimating also how close in an infeasible solution from the feasible region [Zhou 2011]. This is not an easy task, since most real-world problems have complex linear and non-linear constraints, and several approaches have been proposed in the past to handle them [Gen 1997]. From those, the penalty function seems to be yet the most popular
technique for engineering problems, but the intrinsic difficulties to define good penalty values makes harder the optimization process using a GA [Carlos 2000] [Carlos 1999] [Jong1 1993]. In this project, a technique based on the concept of co-evolution is used to create two populations that interact with each other in such a way that the objective function is minimized. The approach has been tested with several single-objective optimization problems with linear and non-linear inequality constraints and its results are compared with those produced by other (GA-based and mathematical programming approaches).

These three approaches (EP, ESs, and GAs) have served to inspire an increasing amount of research on and development of new forms of evolutionary algorithms for use in specific problem solving contexts. A few of these are briefly described below, selected primarily to give the reader a sense of the variety of directions being explored.

One of the most active areas of application of evolutionary algorithms is in solving complex function and combinatorial optimization problems. A variety of features are typically added to EAs in this context to improve both the speed and the precision of the results. A second active area of application of EAs is in the design of robust rule learning systems. One of the open questions here is precisely what changes to EAs need to be made in order to efficiently evolve such complex structures.

Because of the inherent natural parallelism within an EA, much recent work has been concentrated on the implementation of EAs on parallel machines. Typically either one processor holds one individual (in SIMD (Single Instruction Multiple Data) machines), or a subpopulation (in MIMD (Multiple Instruction Multiple Data) machines). Clearly, such implementations hold promise of execution time decreases.
2.3 Message Passing Interface (MPI)

One of the basic methods of programming for parallel computing is the use of message passing libraries. These libraries manage transfer of data between instances of a parallel program running (usually) on multiple processors in a parallel computing architecture.

MPI is a specification for the developers and users of message passing libraries. By itself, it is not a library - but rather the specification of what such a library should be. The goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be practical, portable, efficient and flexible. Interface specifications have been defined for C/C++ and FORTRAN programs.

2.3.1 Reasons for Using MPI:

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.

- **Functionality** - Over 115 routines are defined in MPI-1 alone.

- **Availability** - A variety of implementations are available, both vendor and public domain.
2.3.2 Programming Model:

- MPI lends itself to virtually any distributed memory parallel programming model. In addition, MPI is commonly used to implement (behind the scenes) some shared memory models, such as Data Parallel, on distributed memory architectures.

- Hardware platforms:
  - Distributed Memory: Originally, MPI was targeted for distributed memory systems.
  - Shared Memory: As shared memory systems became more popular, particularly SMP / NUMA architectures, MPI implementations for these platforms appeared.
  - Hybrid: MPI is now used on just about any common parallel architecture including massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks.

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

- The number of tasks dedicated to run a parallel program is static. New tasks cannot be dynamically spawned during run time. (MPI-2 addresses this issue).
2.4 Basic Features of Message Passing Programs

Message passing programs consist of multiple instances of a serial program that communicate by library calls. These calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and finally terminate communications.
2. Calls used to communicate between pairs of processors.
3. Calls that perform communications operations among groups of processors.
4. Calls used to create arbitrary data types.

The first class of calls consists of calls for starting communications, identifying the number of processors being used, creating subgroups of processors, and identifying which processor is running a particular instance of a program.

The second class of calls, called point-to-point communications operations, consists of different types of send and receive operations.

The third class of calls is the collective operations that provide synchronization or certain types of well-defined communications operations among groups of processes and calls that perform communication/calculation operations.

The final class of calls provides flexibility in dealing with complicated data structures.

2.4.1 A Generic MPI Program

All MPI programs have the following general structure:

include MPI header file

variable declarations

initialize the MPI environment

...do computation and MPI communication calls...

close MPI communications
The MPI header file contains MPI-specific definitions and function prototypes. Then, following the variable declarations, each process calls an MPI routine that initializes the message passing environment. All calls to MPI communication routines must come after this initialization.

Finally, before the program ends, each process must call a routine that terminates MPI. No MPI routines may be called after the termination routine is called. Note that if any process does not reach this point during execution, the program will appear to hang.

2.4.2 Communicators

- A **communicator** is a handle representing a group of processors that can communicate with one another.
- The **communicator name** is required as an argument to all point-to-point and collective operations.
- The communicator specified in the send and receive calls must agree for communication to take place.
- Processors can communicate only if they share a communicator.
- There can be many communicators, and a given processor can be a member of a number of different communicators. Within each communicator, processors are numbered consecutively (starting at 0). This identifying number is known as the **rank** of the processor in that communicator.
- The rank is also used to specify the source and destination in send and receive calls.
- If a processor belongs to more than one communicator, its rank in each can (and usually will) be different.
• MPI automatically provides a basic communicator called MPI_COMM_WORLD. It is the communicator consisting of all processors. Using MPI_COMM_WORLD, every processor can communicate with every other processor. You can define additional communicators consisting of subsets of the available processors.

2.4.3 Basic MPI Routines

The following is a collection of some MPI routines.

Table 1: Table showing the basic MPI routines

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int MPI_Init(int *argc, char **argv[])</td>
<td>Initializes MPI environment</td>
</tr>
<tr>
<td>int MPI_Finalize(void)</td>
<td>Terminates MPI execution environment</td>
</tr>
<tr>
<td>int MPI_Comm_rank(MPI_Comm comm, int *rank)</td>
<td>Determines rank of process in communicator</td>
</tr>
<tr>
<td>int MPI_Comm_size(MPI_Comm comm, int *size)</td>
<td>Determines size of group associated with communicator</td>
</tr>
<tr>
<td>double MPI_Wtime(void)</td>
<td>Returns elapsed time from some point in past, in seconds</td>
</tr>
<tr>
<td>int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)</td>
<td>Sends message (blocking)</td>
</tr>
<tr>
<td>int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)</td>
<td>Receives message (blocking)</td>
</tr>
<tr>
<td>int MPI_BARRIER(MPI_Comm comm)</td>
<td>Blocks process until all processes have called it.</td>
</tr>
<tr>
<td>int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)</td>
<td>Broadcasts message from root process to all processes in comm and itself.</td>
</tr>
</tbody>
</table>
3. NARRATIVE

The project mainly focuses on benchmark problems from literature; using constrained functions and solving them for finding the global optimum and the minimum value for the functions. A comparison with other approaches is also provided. The comparison is in terms of the accuracy of the optimum point and the minimal value for the function.

In this project, the two-population evolutionary algorithm [Simionescu 2004] is implemented using C++ and MPI. It uses Manager-Worker Paradigm wherein the manager is the coordinator and the workers run in parallel implementing the function calls etc.

The program uses values for number of female individuals nF, number of male individuals nM, number of iterations nG for which the best female would be constant, and the number of variables in the function.

Tables showing values for the functions for different number of worker processes with a specified number for female and male individuals, iterations are provided for each problem. Graphs are also provided for each problem showing how the computation time is reduced as the number of worker processes is increased.

Evaluation of the test functions using different values for female and male individuals, number of iterations is also provided which shows the accuracy for the minimal values of the functions obtained using this project.
4. PROJECT DESIGN

The goal of the project is to minimize a function $F(X)$ of multiple variables subject to some given constraints using parallelization and the two population algorithm described in the next section. This algorithm is being implemented using C++ programming using MPI in Unix environment. The first implementations will be tested using functions of two variables, and then functions of multiple variables will be used.

4.1 Two Population Evolutionary Algorithm

Implementations of the algorithm are named *female-male evolutionary algorithms*, or $F$-$M(n_F, n_M)$ in short, where $n_F$ is the size of the female population and $n_M$ is the size of the male population [Simionescu 2004]. A general structure of the algorithm is given below. (the steps in the square bracketed steps can be omitted as mentioned in [Simionescu 2004], but during implementation they were used).

**Step 1)** Generate the initial female population of $n_F$ individuals and the male population of $n_M$ individuals; *repeat*

**Step 2:** [Rank by fitness the female individuals];

**Step 3:** [Rank the male individuals based on the number of constraints violated];

**Step 4:** [Mutate females];

**Step 5:** [Mutate males];

**Step 6:** Make female-male pairs and generate offspring;

**Step 7:** Select survivors;

*until a certain Stopping Criteria is met.*
The following is a more detailed discussion of the steps introduced above:

**Step 1:** Uniform random points are generated within the extended intervals:

\[
\left[ x_{i_{\text{min}}} - b_L \cdot \Delta x_i, x_{i_{\text{max}}} + b_R \cdot \Delta x_i \right] \quad (1 \leq i \leq n)
\]  

(5)

where

\[
\Delta x_i = (x_{i_{\text{max}}} - x_{i_{\text{min}}}).
\]  

(6)

Coefficients $b_L$ and $b_R$ (with values between 0 and 1) control the amount in which the infeasible space is expanded so that a male population can be generated and evolved, even when only side constraints are imposed to the optimization problem. If additional constraints are present, the infeasible region may be sufficiently large and the respective coefficients can be set to smaller values, including zero. If desired, different $b_L$ and $b_R$ coefficients can be assigned for different variables $x_i$, as well as extending only the lower boundaries or only the upper boundaries from their initial values $x_{i_{\text{min}}}$ and $x_{i_{\text{max}}}$. Evidently, when the objective function is evaluated, the side constraints are verified as they were posed in the original problem.

If the side constraints are mentioned for an optimization problem, then those boundary conditions can be used while generating the random points, if not, then the user can impose the boundary conditions and generate the random points within those intervals.

If an individual that is randomly generated is in the feasible region of the design space (i.e., the individual satisfies all the constraints), then it will be assigned to the female population, else will be assigned to the male population. The process continues until $n_F$ females and $n_M$ males are generated.
As the program iterates towards generating the initial female and male populations, further useful information about the problem at hand can be acquired: For example, the ratio of the size of the feasible-space (in other words, female population) over the total size of the search space can be estimated. This ratio, noted $\rho$, was determined experimentally in [Michalewicz 1996], prior to solving the optimization problem, by generating a large number of uniform random points inside the search space and counting the number of occurrences of feasible points from the total number of points generated. Knowledge about this ratio can be used in female-male crossover schemes, or in dynamically adjusting the upper and lower limits of the search space, so that the probabilities of randomly generating female and male individuals are balanced.

**Steps 2-7.**

**Step2 (optional step):** The female individuals are ranked based on their fitness, using complete sorting, and the best-fit female (called $\alpha$-female) is identified.

**Step3 (optional step):** Rank the male individuals based on the number of constraints they violate.

**Step4 (optional step):** Mutate females by replacing [the worst ranked] $pF$ % females with randomly generated new females. If female ranking was performed, the newly created females can take the rank of the females they replace (rank inheriting), or Step 2 can be applied one more time until every female has a rank of her own. If no female ranking is performed, then females are mutated at random with, or without preserving the $\alpha$-female, like in an elitist algorithm.

**Step5 (optional step):** Mutate males (the same as it was done with the female population), with a mutation rate of $pM$ %, with or without preserving the $\alpha$-male (the
male that mated during the previous step with the $\alpha$-female). If no male ranking was performed, simply mutate at random $pM\%$ of the male population.

**Step 6**: Form female-male pairs by assigning [at least] one male to each female. If females ranking was applied, then females can choose their mates in a rank decreasing order. In some implementations further called *monogamous-male algorithms*, once a male has been assigned to a female, the respective male will not be available for further mating during the same generation. In other implementations called *polygamous-males algorithms*, males are permitted to recombine with more than one female. It is obvious that for *monogamous-male algorithms* with $nF>nM$, there will be some females that will not participate in the crossover operation, while for *polygamous-male algorithms* this can be avoided. Implementations of the functions in this project are done using *polygamous-male algorithms*.

*Polygamous-male algorithms* can be *unrestricted* i.e. a male can crossover with any number of females, or *restricted* when the number of crossovers a male can perform in one generation is limited to a fraction of the total female population.

After female-male pairs are formed, offspring are generated using one of the known crossover schemes [Adewuya 1993]. Midpoint crossover scheme is used in this project. The offspring can result inside the feasible space (viz. they are females) or outside the feasible space (viz. they are males).

**Step 7**: For convenience, the selection step should be performed as offspring are generated rather than a distinct step, following one of the rules: If the child results outside the feasible space, then he may or may not replaces his father. If the child is a female, she replaces her mother either unconditionally or only if the child has a better fitness than that of the mother. In other implementations of the *polygamous-male*
type, only the mother is replaced, either unconditionally or only if there is a fitness improvement between mother and child.

**Stopping Criteria:** Steps 2 through 7 are repeated until an imposed condition is satisfied; this can be exceeding a maximum number of function calls, attaining a given threshold fitness, or recording the same $\alpha$-female over a given number of generations.

Recording the same $\alpha$-female for a given number of generations is the stopping criteria that is used in this project.
4.2 Parallelization

Since evolutionary algorithms are characterized by their repeated fitness evaluation of the individuals in the population, it is natural to view them as parallel algorithms. In generational evolutionary algorithms, substantial savings in elapsed time can often be obtained by performing fitness evaluations in parallel. In the simplest form of parallelism, a master process performs all the function of the evolutionary algorithm except evaluation of individuals, which are performed in parallel by worker processes operating on separate processors. The master process waits for all workers to return the evaluated individuals before varying on with the next generation.

4.2.1 Parallelization Using Manager-Worker Paradigm

The algorithm is implemented using MPI in which there are \( n \) number of processes that work simultaneously using inter-process communication. This performs well when a complex function is being used and there are more number of iterations that are to be performed.

Manager-Worker paradigm is used to implement the algorithm. In this, there is one Manager and several workers. The Manager generates uniform random points and sends these points to the workers; the workers process these points (check the feasibility and compute the value of the function at that point if it is a feasible point) and send back the processed results. The manager would then differentiate them into female and male based on their feasibility value (female if it is zero and male if not). Then the manager ranks the female individuals based on their fitness and the male individuals based on the number of constraints they violate. Generations of new individuals and mutating female and male individuals would also happen. The female-male pairs are formed with the females choosing their males in the rank
decreasing order. Then the crossover is performed and all the steps are until the best female is constant for a given number of individuals.

Figure 1: Manager-Worker Paradigm
5. TEST FUNCTIONS

This project is tested using various benchmark problems from literature. This section provides the testing of four benchmark problems and their evaluations for different runs that are performed and also comparisons with the previous work.

5.1 Rosenbrock’s Function (in two variables)

In mathematical optimization, the Rosenbrock function is a non-convex function used as a performance test problem for optimization algorithms introduced by Rosenbrock (1960). It is also known as Rosenbrock’s valley or Rosenbrock’s banana function. The global minimum is inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial. To converge to the global minimum however is difficult.

The original Rosenbrock function was unconstrained. The constraints for this function were introduced by Simionescu 2004. It was tested using this algorithm which was implemented in Pascal. The same function using the constraints is tested using this project here.

It is stated as

\[
\text{Minimize } F(x_1, x_2) = 100 \cdot (x_2 - x_1^2)^2 + (x_1 - 1)^2
\]  

(7)

Subject to the following constraints:

\[-2 \leq x_1 \leq 2 \quad \text{and} \quad -2 \leq x_2 \leq 2\]  

(8)

\[g_1(x_1, x_2) \leq 0 \quad \text{AND} \quad g_2(x_1, x_2) \leq 0\]  

(9)

With \(g_1\) and \(g_2\) cubic and linear functions as follows:

\[g_1(x_1, x_2) = (x_1 - 1)^3 - x_2 + 1\]  

\[g_2(x_1, x_2) = x_1 + x_2 - 2\]  

(10)
Figure 2: Plot of Rosenbrock function (in two variables) with constraints

Table 2: Table showing values for Rosenbrock’s Function for different number of processes

<table>
<thead>
<tr>
<th>No. of processes</th>
<th>No. of female individuals</th>
<th>No. of male individuals</th>
<th>No. of iterations</th>
<th>Function Value</th>
<th>Total Time</th>
<th>Optimum Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Algorithm time</td>
<td>Function time</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.268495</td>
<td>2.211688</td>
<td>0.3465099</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.029471</td>
<td>1.351219</td>
<td>0.461643</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.217387</td>
<td>1.224079</td>
<td>0.536698</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.046619</td>
<td>1.11176</td>
<td>0.535684</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.014651</td>
<td>0.99926</td>
<td>0.393892</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.041042</td>
<td>0.996646</td>
<td>0.495492</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.009716</td>
<td>0.971562</td>
<td>0.502946</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>0.007312</td>
<td>0.949841</td>
<td>0.403412</td>
</tr>
</tbody>
</table>
The table 2 above shows the average value for 50 runs with a particular number of processes is given (actually performed the run for up to 20 processes, the table shows only up to 8 processes). The table shows the number of processes (in other words, no. of workers), the female and male population, the number of iterations (for which the best female would be constant), the total time for the run (algorithm time plus function value computation time), the number of functions calls and the minimum point for the function.

![Rosenbrock](image)

Figure 3: Number of processes vs. Time for Algorithm for Rosenbrock’s function

Figure 3 shows the graph for the number of processes vs. the time for algorithm computed at the manager process. Figure 4 shows the graph for the number of processes vs. the total time (time for algorithm plus the function value computation time at the worker processes).

It can be seen from the graphs that as the number of processes increases, the computation time decreases. This is not true for all the values though. This is
because of the number of function calls that make the computation time to rise. If there are more number of function calls, then the time for computing the function value increases, which increases the time for the worker’s response to the manager, which increases the algorithm time and in turn the total time. This is clearly shown with bars for all the times in a single graph.

Figure 4: Number of processes vs. Total Time for Rosenbrock’s function
Figure 5: Single graph showing all the computation times for different number of processes for Rosenbrock’s function

The graph below shows the optimum point for the Rosenbrock function. Theoretical value for the optimum minimum for the function is \((1,1)\). Most of the optimum points computed using various female & male individuals and different number of processes fall between \((0.995, 1.001)\) and \((0.999, 1.004)\) as can be seen from the graph. The best optimum point that is found using this algorithm is \((0.999376, 1.00052)\) at which the function value is 0.000312.
Figure 6: Graph showing optimum point for Rosenbrock’s function
5.2 Benacer-Tao Function (in two variables)

It was introduced by Benacer, Tao 1986. The function can be stated as:

\[
\text{Minimize } f(x_1, x_2) = x_1 - \frac{1}{2}x_2 - \frac{2}{3}x_1^2 + \frac{1}{2}x_2^2
\]  

(11)

Subject to:

\[
g(x_1, x_2) = 2x_1 - x_2 - 3 \leq 0
\]

\[
x_1 \geq 0
\]

\[
x_2 \geq 0
\]  

(12)

Figure 7: Plot of Benacer-Tao function

Following is a table in which the average value for 50 runs with a particular number of processes is performed. The table shows the number of processes (in other words, no. of workers), the female and male population, the number of iterations (for which the best female would be constant), the total time for the run (algorithm time plus function value computation time), the number of function calls and the optimum point for the function.
The table shows it only for up to 8 processes, the experiment is performed for up to 20 processes. The graph below is the one which is drawn using the number of processes and the total required for the completion of the algorithm.

Table 3: Table showing the values for different number of processes for BenacerTao function

<table>
<thead>
<tr>
<th>No. of processes</th>
<th>No. of female points</th>
<th>No. of male points</th>
<th>No. of iterations</th>
<th>Function Value</th>
<th>Total Time</th>
<th>No. of function calls</th>
<th>Optimum Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74875</td>
<td>0.449998</td>
<td>5470</td>
<td>(2.25818, 1.5175)</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74894</td>
<td>0.38384</td>
<td>6601</td>
<td>(2.24248, 1.48597)</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74999</td>
<td>0.228379</td>
<td>5438</td>
<td>(2.25188, 1.50376)</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74930</td>
<td>0.299412</td>
<td>6432</td>
<td>(2.27155, 1.54318)</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74924</td>
<td>0.239412</td>
<td>5576</td>
<td>(2.22644, 1.45289)</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74949</td>
<td>0.225073</td>
<td>5696</td>
<td>(2.2387, 1.47775)</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74855</td>
<td>0.217929</td>
<td>5407</td>
<td>(2.25951, 1.52032)</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-0.74913</td>
<td>0.31041</td>
<td>6595</td>
<td>(2.23128, 1.49712)</td>
</tr>
</tbody>
</table>
Figure 8: Number of processes vs. Time for Algorithm for Benacer-Tao function

Figure 8 shows the graph for the number of processes vs. the time for algorithm computed at the manager process. Figure 8 shows the graph for the number of processes vs. the total time (time for algorithm plus the function value computation time at the worker processes). From the graph, it can be seen that as the number of processes increases, the time decreases. The graph is not a completely decreasing one, as the total time is dependent on the function computation time calculated at the worker processes (also depends on the number of function calls). Figure 9 clearly shows all the computation times in a single graph which shows that the time for algorithm is increased whenever there is an increase in the function value computation time (which depends on the number of function calls).
Figure 9: Number of Processes vs. Total Time graph for Benacer-Tao function

Figure 10: Single graph showing all the computation times for different number of processes for Benacer-Tao function
Figure 11: Graph showing the optimum point for Benacer-Tao function

The graph above shows the optimum point for the Benacer-Tao function. Theoretical value for the optimum minimum for the function is (2.25, 1.5) at which the function value is -0.75. Most of the optimum values computed using this algorithm for various values of nF, nM, nG and the number of processes, fall between (2.248906, 1.49968) and (2.25176, 1.5175). The best optimum point found is (2.24999, 1.4999) at where the function value is -0.75.
5.3 Rosen-Suzuki Fist Function (in four variables)

The Rosen-Suzuki problem is a function of four variables with three nonlinear constraints on the variables. It is taken from problem 43 of Hock and Schittkowski (1981). The objective function is

\[ f(x) = x_1^2 + x_2^2 + 2x_3^2 + x_4^2 - 5x_1 - 5x_2 - 21x_3 + 7x_4 \]  (13)

The nonlinear constraints are

\[
\begin{align*}
0 & \leq 8 - x_1^2 - x_2^2 - x_3^2 - x_4^2 - x_1 + x_2 - x_3 + x_4 \\
0 & \leq 10 - x_1^2 - 2x_2^2 - x_3^2 - 2x_4^2 + x_1 + x_4 \\
0 & \leq 5 - 2x_1^2 - x_2^2 - x_3^2 - 2x_1 + x_2 + x_4
\end{align*}
\]  (14)

The theoretical value for the optimum minimum for the function is -44 which is located at (0,1,2,-1). The best value that is obtained using this project is -43.9756 at (-0.00021, 0.99821, 2.00281, -0.98912). The best value that is reported in the literature is (-0.0005463221, 1.000618, 2.000213, -0.9996195) at which the optimum value for the function is -44 [Adewuya 1993], this was using the GaNOP system.

Following is a table in which the average value for 50 runs with a particular number of processes is performed. The table shows the number of processes (in other words, no. of workers), the female and male population, the number of iterations (for which the best female would be constant), the total time for the run (algorithm time plus function value computation time), the number of function calls and the optimum point for the function. The table shows it only for up to 8 processes, the experiment is performed for up to 20 processes. The graph below is the one which is drawn using the number of processes and the total required for the completion of the algorithm.
Table 4: Table showing the values for different number of processes for Rosen-Suzuki function

<table>
<thead>
<tr>
<th>No. of processes</th>
<th>No. of female points</th>
<th>No. of male points</th>
<th>No. of iterations</th>
<th>Function Value</th>
<th>Total Time</th>
<th>No. of function calls</th>
<th>Optimum Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-43.0923</td>
<td>11.37833</td>
<td>11571</td>
<td>(-0.04239, 0.89619, 1.9886, -0.955717)</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-43.9503</td>
<td>9.514633</td>
<td>11578</td>
<td>(-0.000138, 0.97898, 2.006346, -0.986454)</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-43.9362</td>
<td>9.362131</td>
<td>11598</td>
<td>(-0.002435, 0.97898, 2.002316, -0.996354)</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-43.7521</td>
<td>8.182058</td>
<td>11611</td>
<td>(-0.01134, 0.87698, 2.02318, -0.97867)</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-43.2259</td>
<td>9.039228</td>
<td>11620</td>
<td>(-0.00054, 0.984798, 1.95489, -0.973141)</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-42.6588</td>
<td>8.117775</td>
<td>11635</td>
<td>(-0.00187, 0.96715, 1.98712, -0.98715)</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-43.5648</td>
<td>9.82167</td>
<td>11635</td>
<td>(-0.00583, 0.984923, 2.0182, -0.88342)</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>15</td>
<td>1000</td>
<td>-43.872</td>
<td>10.11201</td>
<td>14321</td>
<td>(-0.00034, 0.98672, 2.00178, -0.97821)</td>
</tr>
</tbody>
</table>

![Rosen-Suzuki Fist Function](image_url)

**Rosen-Suzuki Fist Function**

- Time for Algorithm
Figure 12: Number of processes vs. Time for Algorithm for Rosen-Suzuki Fist function

Figure 12 shows the graph for the number of processes vs. the time for algorithm computed at the manager process. Figure 12 shows the graph for the number of processes vs. the total time (time for algorithm plus the function value computation time at the worker processes). From the graph, it can be seen that as the number of processes increases, the time decreases. The graph is not a completely decreasing one, as the total time is dependent on the function computation time calculated at the worker processes (also depends on the number of function calls). Figure 14 clearly shows all the computation times in a single graph which shows that the time for algorithm is increased whenever there is an increase in the function value computation time (which depends on the number of function calls).

![Rosen-Suzuki Fist Function](image)

**Figure 13**: Number of processes vs. Total time graph for Rosen-Suzuki Fist function
Figure 14: Single graph showing all the computation times for different number of processes for Rosen-Suzuki function
5.4 Himmelblau Nonlinear Optimization Problem (in five variables)

This problem was originally proposed by Himmelblau 1972, and it was chosen to try this algorithm as it was used before as a benchmark for several other GA-based techniques. In this problem, there are five design variables \((x_1, x_2, x_3, x_4, x_5)\), 6 nonlinear inequality constraints and ten boundary conditions. The problem can be stated as follows:

\[
\text{Minimize } f(X) = 5.3578547x_3^2 + 0.8356891x_1x_5 + 37.29329x_1 - 40792.141
\]

Subject to :

\[
g_1(X) = 85.334407 + 0.0056858x_2x_5 + 0.00026x_1x_4 - 0.0022053x_3x_5 \\
g_2(X) = 80.51249 + 0.0071317x_2x_5 + 0.0029955x_1x_2 + 0.0021813x_3^2 \\
g_3(X) = 9.300961 + 0.0047026x_3x_5 + 0.0012547x_1x_3 + 0.0019085x_3x_4
\]

\[
0 \leq g_1(X) \leq 92 \\
90 \leq g_2(X) \leq 110 \\
20 \leq g_3(X) \leq 25 \\
78 \leq x_1 \leq 102 \\
33 \leq x_2 \leq 45 \\
27 \leq x_3 \leq 45 \\
27 \leq x_4 \leq 45 \\
27 \leq x_5 \leq 45
\]

5.4.1 Comparison of results:

This problem was originally proposed by Himmelblau and solved using the Generalized Reduced Gradient method (GRG). Gen and Cheng solved this problem using a genetic algorithm based on both local and global reference. The result shown in Table 4 is the best found with their approach.

Homaifar, Qi and Lai solved this problem using a genetic algorithm with a population size of 400, and their results were the previously best reported.

The solution shown for the algorithm given here is the best produced after 50 runs and using the following ranges for the design variables: \(78.000 \leq x_1 \leq 102.000\),
33.0000 ≤ x_2 ≤ 45.0000, \quad 27.0000 ≤ x_3 ≤ 45.0000, \quad 27.0000 ≤ x_4 ≤ 45.0000, \quad 27.0000 ≤ x_5 ≤ 45.0000. The values from x_1 to x_5 are considered with a 4-decimal precision.

The mean for the 100 runs performed is f(X) = -30786.5. The worst solution found is f(X) = -30692.9, which is better than the best solution previously reported. The best solution found with this algorithm is f(X) = -30868.9 (corresponding to x_1 = 78.7032, x_2 = 33.7124, x_3 = 27.83889, x_4 = 44.2504 and x_5 = 43.1159).

There were 20 female and 15 male points generated in each iteration. This was run using 8 processing units in which the best solution remained constant for 1000 iterations. The duration for the run is 0.419021 sec. It was run using 8 processing units. The number of function calls is 59585.

Table 5: Comparison of current algorithm with previous techniques for Himmelblau Problem

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1</td>
<td>78.7032</td>
<td>81.4900</td>
<td>78.0000</td>
<td>78.6200</td>
</tr>
<tr>
<td>x_2</td>
<td>33.7124</td>
<td>34.0900</td>
<td>33.00000</td>
<td>33.4400</td>
</tr>
<tr>
<td>x_3</td>
<td>27.8388</td>
<td>31.2400</td>
<td>29.9950</td>
<td>31.0700</td>
</tr>
<tr>
<td>x_4</td>
<td>44.2504</td>
<td>42.2000</td>
<td>45.0000</td>
<td>44.1800</td>
</tr>
<tr>
<td>x_5</td>
<td>43.1159</td>
<td>34.3700</td>
<td>36.7760</td>
<td>35.2200</td>
</tr>
<tr>
<td>g_1(X)</td>
<td>91.8574</td>
<td>90.522543</td>
<td>90.714681</td>
<td>90.520761</td>
</tr>
<tr>
<td>g_2(X)</td>
<td>100.517</td>
<td>99.318806</td>
<td>98.840511</td>
<td>98.892933</td>
</tr>
<tr>
<td>g_3(X)</td>
<td>20.0456</td>
<td>20.060410</td>
<td>19.999935</td>
<td>20.131578</td>
</tr>
<tr>
<td>f(X)</td>
<td>-30868.9</td>
<td>-30183.576</td>
<td>-30665.609</td>
<td>-30373.949</td>
</tr>
</tbody>
</table>
6. SUMMARY OF RESULTS

The table below shows the evaluation of the above four benchmark problems. The function values shown are taken average for 20 runs using 8 processes. Each function is run using the given values for number of female individuals, number of male individuals and the number of iterations for which the function value would be constant (at a minimum). The minimum point is not mentioned, only the minimum value for the function is mentioned to maintain uniformity with Himmelblau (where the question is to find the lowest function value, but not the global minimum).

nF – number of female points
nM – number of male points
nG – number of generations (iterations)

Table 6: Evaluation of functions for different nF, nM, nG values

<table>
<thead>
<tr>
<th>nF, nM, nG values</th>
<th>Function</th>
<th>No. of function calls</th>
<th>Optimum Point</th>
<th>Function Value</th>
<th>Fbest</th>
<th>Fworst</th>
</tr>
</thead>
<tbody>
<tr>
<td>nF=20, nM=15, nG=1000</td>
<td>Rosenbrock</td>
<td>3460</td>
<td>(0.983635,1.00829)</td>
<td>0.16324</td>
<td>0.0162782 at (0.994253,1.00128)</td>
<td>0.206257 at (0.984546,1.01472)</td>
</tr>
<tr>
<td></td>
<td>Benacer-Tao</td>
<td>1117</td>
<td>(2.23332,1.46735)</td>
<td>-0.748936</td>
<td>-0.74992 at 2.24575,1.49155</td>
<td>-0.748581 at 2.22248,1.44539</td>
</tr>
<tr>
<td></td>
<td>Rosen-Suzuki</td>
<td>5354</td>
<td>(-0.01863,0.95885, 1.98897,-0.92999)</td>
<td>-43.2825</td>
<td>-43.9756 at (-0.00021,0.99821,2.00281,-0.98912)</td>
<td>-43.0923 at (-0.04239,0.89619,1.9886,-0.955717)</td>
</tr>
<tr>
<td></td>
<td>Himmelblau</td>
<td>2887</td>
<td>(79.7158,34.5415, 28.43,6472,42.3381)</td>
<td>-30.798.2</td>
<td>-30.842.3 at (78.43,5,4.9001,28.4007,44.2984,41.2387)</td>
<td>-30.692.9 at (78.0102,34.6353,28.6994,44.7706,42.6316)</td>
</tr>
<tr>
<td>nF=50, nM=45, nG=1000</td>
<td>Rosenbrock</td>
<td>3020</td>
<td>(0.987025,1.00648)</td>
<td>0.104249</td>
<td>0.0433724 at (0.989938,1.00078)</td>
<td>0.24715 at (0.977561,1.00529)</td>
</tr>
<tr>
<td></td>
<td>Benacer-Tao</td>
<td>3004</td>
<td>(2.22193,1.44492)</td>
<td>-0.747949</td>
<td>-0.979966 at 2.24646,1.49293</td>
<td>-0.744298 at 2.22993,1.46523</td>
</tr>
<tr>
<td></td>
<td>Rosen-Suzuki</td>
<td>10350</td>
<td>(-0.18876,1.12005,2.04479,-0.877288)</td>
<td>-43.3159</td>
<td>-43.3502 at (-0.244337,0.9509,2.10323,0.894361)</td>
<td>-43.2226 at (-0.22435,0.89450,2.1567,-0.758529)</td>
</tr>
<tr>
<td>Function</td>
<td>nF</td>
<td>nM</td>
<td>nG</td>
<td>f(x)</td>
<td>g(x)</td>
<td>h(x)</td>
</tr>
<tr>
<td>----------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>8564</td>
<td>(80.395,33.6028, 27.4616,43.7525, 44.0002)</td>
<td>-30797.2</td>
<td>(80.0442,34.5086, 27.8807,44.4072, 42.2334)</td>
<td>-30738.6</td>
<td>(78.2194,35.7742, 28.8082,42.1174, 41.151)</td>
</tr>
<tr>
<td>nF=100</td>
<td>nM=100</td>
<td>nG=1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>6467</td>
<td>(0.983635,1.00829)</td>
<td>0.16324</td>
<td>(0.981793,1.00065)</td>
<td>0.24715</td>
<td>(0.97364,1.00029)</td>
</tr>
<tr>
<td>Benacer-Tao</td>
<td>6781</td>
<td>(2.24902,1.49916)</td>
<td>-0.748869</td>
<td>(-2.26972,1.52977)</td>
<td>-0.748017</td>
<td>(-2.23382,1.46932)</td>
</tr>
<tr>
<td>Rosen-Suzuki</td>
<td>21341</td>
<td>(0.02258,1.129959, 1.97909, -0.846198)</td>
<td>-43.419</td>
<td>(0.01015,0.967676, 1.98512,-1.0206)</td>
<td>-43.3299</td>
<td>(-0.11234,0.68412, 2.13743,-0.846198)</td>
</tr>
<tr>
<td>Himmelblau</td>
<td>5642</td>
<td>(78.6626,34.1357, 27.8584,42.4566, 43.5148)</td>
<td>-30839.8</td>
<td>(78.7032,33.7124, 27.8388,44.2504, 43.1159)</td>
<td>-30798.8</td>
<td>(79.2046,35.0102, 28.3194,43.9151, 41.435)</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>11829</td>
<td>(0.985867,1.01213)</td>
<td>0.161779</td>
<td>(0.989938,1.00078)</td>
<td>0.005039</td>
<td>(0.973804,1.02603)</td>
</tr>
<tr>
<td>Benacer-Tao</td>
<td>13127</td>
<td>(2.26972,1.52977)</td>
<td>-0.749129</td>
<td>(-2.24576,1.4917)</td>
<td>-0.748869</td>
<td>(-2.24902,1.49916)</td>
</tr>
<tr>
<td>Rosen-Suzuki</td>
<td>52834</td>
<td>(0.02329,1.04194, 1.91381,-1.08141)</td>
<td>-43.5051</td>
<td>(-0.0987, 1.02341, 1.89921,-1.0721)</td>
<td>-43.1385</td>
<td>(-0.259146,0.87214, 1.7934,-1.22647)</td>
</tr>
<tr>
<td>Himmelblau</td>
<td>16766</td>
<td>(79.5418,34.8963, 28.3511,44.364, 41.281)</td>
<td>-30775.2</td>
<td>(79.1098,33.5566, 27.868,44.4233, 43.7728)</td>
<td>-30746.1</td>
<td>(79.9271,34.5459, 28.265,43.9021, 41.6931)</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>20671</td>
<td>(0.994349,1.00137)</td>
<td>0.015997</td>
<td>0.0130185</td>
<td>0.0252508</td>
<td>(0.993588,1.000309)</td>
</tr>
<tr>
<td>Benacer-Tao</td>
<td>28579</td>
<td>(-0.15157,0.95186, 2.09157,-0.863694)</td>
<td>-0.748444</td>
<td>(-0.749018,2.25593,1.51278)</td>
<td>-0.748017</td>
<td>(-2.23382,1.46932)</td>
</tr>
<tr>
<td>Rosen-Suzuki</td>
<td>87591</td>
<td>(-0.01342,0.93123, 1.98791,-0.99127)</td>
<td>-43.5589</td>
<td>(-0.01587,0.92102, 2.07428,-0.947395)</td>
<td>-43.4933</td>
<td>(-0.086978,1.07306, 1.96718,-0.860105)</td>
</tr>
<tr>
<td>Himmelblau</td>
<td>39230</td>
<td>(79.1098,33.5566, 27.868,44.4233, 43.7728)</td>
<td>-30786.9</td>
<td>(78.4546,35.0215, 28.3487,42.1596, 42.253)</td>
<td>-30726.9</td>
<td>(78.8453,35.9396, 28.8694,44.354, 40.3604)</td>
</tr>
</tbody>
</table>
Solving constraint nonlinear programming problems using a co-evolutionary algorithm was implemented. The two-population evolutionary algorithm proposed by Simionescu 2004 was used in which two distinct populations are evolved. The interaction between these populations induces an exploratory pressure on the boundaries of the feasible space. Searching capabilities inside the feasible space and parallel to the feasible-infeasible boundary were induced by mutating the female and male populations.

There is no penalty factor that is involved in constraint handling, hence a clear distinction can be made between the feasible and infeasible individuals. Parallel Implementation of the Co-evolutionary Genetic Algorithm has been a long awaited development for the evolutionary algorithms as mentioned in [Carlos 1999] [Jong 1993] [Simionescu 2004]. In this project, the algorithm has been implemented using MPI, which uses multiple processes that run in parallel, by which the computation time is reduced. This implementation worked well with several test problems that were previously solved using GA-based and mathematical programming techniques, producing in most of the cases results better than those previously reported. The technique is able to achieve such good results with relatively small populations and using a relatively low number of generations.

Functions of multiple variables used in various disciplines of Engineering can be solved for the global optimum using this project. This project gave good results in terms of the computation time and the number of generations that need to be evolved for a more accurate value. The project is tested using four benchmark problems from
the literature. A detailed comparison of the results obtained from the project with that of those from the literature is presented.
8. FUTURE WORK

This project can be further implemented using other properties and fitness functions for ranking, mutation and crossover. Still better results can be obtained using these.

The project implementation is almost like a sequential algorithm where most of the algorithm is done by the Manager process (generating random points, mutation, ranking, crossover etc.). The worker processes just compute the feasibility and the function value which is done in parallel among all the worker processes. More better results can be obtained if this is implemented using other parallelization techniques where the algorithm and computation can be equally divided among the processes participating in the algorithm.

BLX-α crossover [Eschelman and Schaffer 1993] between females and males (rather than the midpoint crossover as currently experimented with), can also be employed in order to enhance the global searching capabilities of the algorithm. There are other crossover schemes that can be employed such as One-point crossover, Two-point crossover, Uniform crossover [Adewuya 1993], Simple crossover, Heuristic crossover [Michalewicz 1994]. Better results can be obtained using these.

Other techniques for mutation can also be employed such as Uniform Mutation, Non-uniform mutation, boundary mutation [Michalewicz 2004].

Numerous other variants of female-male evolutionary algorithms remain to be implemented and tested, making them very appealing.
ACKNOWLEDGEMENTS

The preparation of this report and completion of the project was successful because of the never ending support and guidance of Dr. Michael Scherger, Associate Professor of the Department of Computing Sciences, Texas A & M University – Corpus Christi.

I would like to express my sincere thanks to Dr. P. A. Simionescu, Texas A&M University – Corpus Christi for his suggestions, comments and guidance throughout the project. His support has tremendously helped to ensure to the success of the project.

I would like to express my sincere thanks to Dr. David Thomas, Associate Professor of Computing Sciences, Texas A&M University – Corpus Christi, for his unending support and warm wishes that helped me to concentrate on completing my project.

My sincere heartfelt thanks to all the faculty, and staff of the Department of Computing Sciences for their outstanding support.

Last but not least, I would like to thank my parents, family who provided the much needed moral support and boosted me in reaching the successful completion of the project.
REFERENCES


• http://www.mcs.anl.gov/research/projects/mpi/tutorial/gropp/talk.html
• http://www.mcs.anl.gov/research/projects/mpi/tutorial/
• http://www.lam-mpi.org/tutorials/nd/
• https://computing.llnl.gov/tutorials/mpi/
• http://www.eece.maine.edu/~zhu/ece498/references/Basic%20MPI%20Routines.pdf
APPENDICES

1. main.cpp

// **********************************************************************
// Programmer : Roja Ramani Molupoju
// Date Last Updated : 27th April, 2011
// **********************************************************************

// includes and defines

#include<iostream>
#include<mpi.h>
#include<cstdlib>
#include<cmath>
#include "functions.h" // header file in which all the test functions are written

using namespace std;
#define WORKTAG 1
#define DIETAG 2
#define Iterations 1000 // No. of iterations (generations)

// global variables

const int nF = 100, nM = 100; // Number of female and male individuals
const int n = 2; // number of variables in the function
int rank, fcount = 0, mcount = 0; // rank of process, female individual count and male individual count
double Point[n], work[n]; // array of double values in which random points are stored
int mutatecount = 50; // mutation count

double start_func, end_func, tot_func = 0.0; // variables for calculating the duration for computing function value
int num_func_call = 0; // variable for knowing the number of function calls

// structure for the randomly generated points
struct RandomPoint
{
    double p[n];       // the coordinates are stored
    int feasible;      // feasibility of the point
    double func_value; // function value
} FemaleInd[nF], MaleInd[nM], result;

// ************************************************************
// // Local Functions
// // ********************************************************
void manager(void); // called for Process 0
void worker(void);  // called for other processes
void GenerateRandomPoints(int ntasks); // generates nF female and nM male
random points
void get_next_work_item(double work[]); // generates a random point within
specified interval
void process_results(RandomPoint result); // processes the results obtained from
the workers
RandomPoint do_work(double work[]); // takes a random point, checks the
feasibility and computes function value
void GenerateOffspring(struct RandomPoint &f, struct RandomPoint &m); //
generates an offspring
bool Compare(struct RandomPoint a, struct RandomPoint b); // compares two
random points
void Female_Sort(); // sorts female point
void Male_Sort();  // sorts male points

//***************************************************************
// // main function
// //***************************************************************
int main(int argc, char *argv[])
{
    // Initialize MPI
    MPI_Init(&argc, &argv);

    srand(time(NULL));

    // Find the identity in the default communicator
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0)
        manager();
    else
        worker();

    // Shut down MPI
MPI_Finalize();
return 0;
}
// ***************************************************************
//
// manager function
//
// Called for process 0. It generates random points and sends it to the workers.
// After receiving from workers, it divides the points into male and female,
// mutates the points, sorts them, forms female-male pairs and generates
// offspring. Find the best female in each iteration and compares it with the
// previous best female. If the best female stays constant for specified number of
// iterations, then it sends quit to worker processes which in turn send the
// function time and no. of function calls and exit
//
// ***************************************************************

void manager(void)
{
    int ntasks, pid, ind, gen=0;

    MPI_Status status;

    double start_time, end_time, start_mutate, end_mutate, start_sort, end_sort, 
    start_cross, end_cross, total_time = 0.0, total_mutate = 0.0, total_sort = 0.0, 
    total_cross = 0.0;
    double total_func = 0.0;
    // Size of the communicator
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    start_time = MPI_Wtime();

    // Generate random points and send to workers
    GenerateRandomPoints(ntasks);

    // printing out the number of female and male individuals generated
    cout<<"Number of female individuals : "<<fcount<<"\n";
    cout<<"Number of male individuals : "<<mcount<<"\n";

    int cnt, j, counter = 0;

    start_sort = MPI_Wtime();
    Female_Sort(); // sorting female individuals

    Male_Sort(); // sorting male individuals
    end_sort = MPI_Wtime();

    total_sort += (end_sort - start_sort);
RandomPoint best_female;
    best_female = FemaleInd[0]; // assigning the best female

do
{
    int i, j;
    RandomPoint temp;

    fcount = mcount = mutatcount; // assigning the mutate count

    start_mutate = MPI_Wtime();
    // Mutating
    GenerateRandomPoints(ntasks); // mutating
    end_mutate = MPI_Wtime();

    total_mutate += (end_mutate - start_mutate);

    start_sort = MPI_Wtime();
    // Sorting
    Female_Sort();
    Male_Sort();
    end_sort = MPI_Wtime();

    total_sort += (end_sort - start_sort);

    RandomPoint prev_best ;
    prev_best = best_female; // assigning previous best

    start_cross = MPI_Wtime();
    // Cross-over
    int b;
    for(int a=0; a<nF; a++)
    {
        if(a > nM)
            b = a-nM;
        else b = nM - a;
        GenerateOffspring(FemaleInd[a], MaleInd[b]); // Generating offsprings
    }
    end_cross = MPI_Wtime();

    total_cross += (end_cross - start_cross);

    best_female = FemaleInd[0];

    // Comparing previous and current best females
    bool comp = Compare(prev_best,best_female);

    if(comp)
{ 
counter++;
} 
else 
{
gen ++;

cout<"prev best & best female ";
for(j=0; j<n; j++)
cout<prev_best.p[j]<<"."

cout<prev_best.func_value<<" & ";
for(j=0; j<n; j++)
cout<best_female.p[j]<<"."

cout<best_female.func_value<<endl;
counter=0;
}
}while(counter < Iterations);

// There is no more work to be done, so receive all the outstanding results from the workers
for(rank = 1; rank < ntasks; rank++)
{
    MPI_Recv(&result, sizeof(RandomPoint), MPI_BYTE,
    MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
}

// Send exit message to all the workers with the DIETAG
for(rank = 1; rank < ntasks; rank++)
    MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);

num_func_call = 0;
int nfc; // number of function calls to be received from workers
for(rank = 1; rank < ntasks; rank++)
{
    MPI_Recv(&tot_func, 1, MPI_DOUBLE, rank, DIETAG,
    MPI_COMM_WORLD, &status); // receiving the function computation time from each worker
    MPI_Recv(&nfc, 1, MPI_INT, rank, DIETAG, MPI_COMM_WORLD, &status); // receiving number of function calls from each worker
    total_func += tot_func;
    num_func_call += nfc;
}
end_time = MPI_Wtime();

int i;
cout<" best female :";
for(i=0; i<n ; i++)
cout<<best_female.p[i]<<",";
cout<<best_female.func_value<<endl;

    total_time = end_time - start_time;
    // displaying
    cout<<" Duration \t Function \t #FunctionCalls \t Mutation \t Sorting \t Cross-Over \n"<<endl;
    printf("%f \t %f \t %d \t %f \t %f \t %f\n", total_time, total_func,
num_func_call/(gen), total_mutate, total_sort, total_cross);
    }
}

void worker(void)
{
    MPI_Status status;
    double work[n];
    RandomPoint result;

    while(1)
    {
        // Receive a message from the manager
        MPI_Recv(&work, n, MPI_DOUBLE, 0, MPI_ANY_TAG,
MPI_COMM_WORLD, & status);

        // check the tag of the received message
        if(status.MPI_TAG == DIETAG)
        {
            MPI_Send(&tot_func, 1, MPI_DOUBLE, 0, DIETAG,
MPI_COMM_WORLD);
            MPI_Send(&num_func_call, 1, MPI_INT, 0, DIETAG,
MPI_COMM_WORLD);
            return;
        }

        // Do the work
        result = do_work(work);

        // send the result back
        MPI_Send(&result, sizeof(RandomPoint), MPI_BYTE, 0, 0,
MPI_COMM_WORLD);
    }
}
void GenerateRandomPoints(int ntasks)
{
    int pid;
    MPI_Status status;
    // fcount = mcount = 10;

    for(pid =1; pid < ntasks; pid++)
    {
        // Find the next item of work to do
        get_next_work_item(work);

        // send the work to each worker
        MPI_Send(&work, n, MPI_DOUBLE, pid, WORKTAG, MPI_COMM_WORLD);
    }

    // Loop over getting new work requests until there is no more work to be done
    get_next_work_item(work);

    while(fcount < nF || mcount < nM)
    {
        // Receive results from a worker
        MPI_Recv(&result, sizeof(RandomPoint), MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        pid = status.MPI_SOURCE;
        process_results(result);

        // send the worker a new work unit
        MPI_Send(&work, n, MPI_DOUBLE, pid, WORKTAG, MPI_COMM_WORLD);

        // Get the next work unit to be done
        get_next_work_item(work);
    }
}
void get_next_work_item(double *work)
{
    int i;
    int xmin = -10, xmax = 10;
    int xdef = xmax - xmin;
    float bl = 0.1, br=0.1;

    // for Rosenbrock
    work[0] = drand48()*4.0 - 2;
    work[1] = drand48()*4.0 - 1;

    /* // for Benacer-Tao and Rosen-Suzuki
    for(i=0; i<n; i++)
    {
        work[i] = drand48()*5.0;// - 5;
    }
    */
    /* // for SpeedReducer Problem
    work[0] = (drand48() + 2.6);
    work[1] = (drand48() + 7.0)/10;
    work[2] = drand48() * 17.0 + 6.0;
    work[3] = work[4] = drand48() + 7.3 ; //(drand48() + 7.3);
    work[5] = (drand48() + 2.9);
    work[6] = drand48() + 5.0;
    */
    /*// For Himmelblau Problem
    work[0] = drand48()*24 + 78;
    work[1] = drand48()*12 + 33;
    work[2] = drand48()*18 + 27;
    work[3] = drand48()*18 + 27;
    work[4] = drand48()*18 + 27;
    */
    /* // For Problem4
    work[0] = (rand()%99 + 1)*0.0625;
    work[1] = (rand()%99 + 1)*0.0625;
    work[2] = drand48()*200.0;
    work[3] = drand48()*200.0;
    */
    /* // Benchmark Problem4
    for(i=0; i<9; i++)
    work[i] = drand48();
*/
for(i=9; i<12; i++)
work[i] = (double)(rand()%100);

work[12] = drand48();/*
/*
work[0] = drand48()*900 + 100;
work[1] = drand48()*9000 + 1000;
work[2] = drand48()*9000 + 1000;
work[3] = drand48()*990 + 10;
work[4] = drand48()*990 + 10;
work[5] = drand48()*990 + 10;
work[6] = drand48()*990 + 10;
work[7] = drand48()*990 + 10;

*/
}

//**************************************************************************
//  process_results function
//**************************************************************************

void process_results(struct RandomPoint result)
{
    int i;

    if(result.feasible == 0 && fcount < nF)
    {
        // if(!CheckPoint(result, result.feasible))
        // {
        FemaleInd[fcount] = result;
        fcount++;
        // cout<<"female count"<<fcount<<endl;
        //    }
    }
    else
    {
        if(!result.feasible == 0) && mcount < nM)
        {
            // if(!CheckPoint(result, result.feasible))
            // {
            MaleInd[mcount] = result;
            mcount++;
            //  }
        }
    }
}
RandomPoint do_work(double work[]) {
    RandomPoint Individual;
    int feasible;
    double funcvalue;
    int i;

    for(i=0; i<n; i++)
        Individual.p[i] = work[i];

    start_func = MPI_Wtime();
    RosenbrockValue(work, feasible, funcvalue, n, num_func_call);
    end_func = MPI_Wtime();

    tot_func += end_func - start_func;

    Individual.feasible = feasible;
    Individual.func_value = funcvalue;

    return Individual;
}

void GenerateOffspring(struct RandomPoint &f, struct RandomPoint &m) {
    RandomPoint c;
    int feasible;
    double funcvalue;
    int t;

    for(t=0; t<n; t++)
        c.p[t] = (f.p[t] + m.p[t])/2;
RosenbrockValue(c.p, feasible, funcvalue, n, num_func_call);

c.feasible = feasible;
c.func_value = funcvalue;

if(feasible == 0)
{
    if(c.func_value < f.func_value)
        f = c;
} else
    m = c;

//*********************************************************************
// Compare function
//*********************************************************************

bool Compare(struct RandomPoint a, struct RandomPoint b)
{
    int i, c=0;
    bool comp;

    for(i=0; i<n; i++)
        if(a.p[i] == b.p[i])
            c++;

    if(c==n)
        comp = true;
    else comp=false;

    return comp;
}

//*********************************************************************
// Female_Sort function
//*********************************************************************

void Female_Sort()
{
    RandomPoint temp;
int i, j;

for(i=nF-2; i>=0; i--)
{
    for(j=0; j<=i; j++)
    {
        if(FemaleInd[j].func_value > FemaleInd[j+1].func_value)
        {
            temp = FemaleInd[j];
            FemaleInd[j] = FemaleInd[j+1];
            FemaleInd[j+1] = temp;
        }
    }
}

//*****************************************************************
// Male_Sort function
//****************************************************************

void Male_Sort()
{
    RandomPoint temp;
    int i, j;

    for(i=nM-2; i>=0; i--)
    {
        for(j=0; j<=i; j++)
        {
            if(MaleInd[j].feasible > MaleInd[j+1].feasible)
            {
                temp = MaleInd[j];
                MaleInd[j] = MaleInd[j+1];
                MaleInd[j+1] = temp;
            }
        }
    }
}
# Functions.h

#include<iostream>
#include<stdlib>
#include<cmath>

using namespace std;

#define pi 3.14

//*****************************************************************************
// RosenbrockValue function
//*****************************************************************************

double RosenbrockValue(double x[], int &feasible, double &funcvalue, int n, int &num_func_call)
{
    int feasibility = 0, c1, c2;
    double Constraint1;
    double Constraint2;

    if(x[0] >= -2 && x[0] <= 2)
        c1 = 0;
    if(x[1] >= -1 && x[1] <= 3)
        c2 = 0;

    Constraint1 = pow((x[0] - 1), 3) - x[1] + 1;
    Constraint2 = x[0] + x[1] - 2;

    if( c1 == 0 && c2 == 0)
    {
        if(Constraint1 <= 0 && Constraint2 <= 0)
            feasibility = 0;
        else
            feasibility = 1;
    }
    else
    {
        if(c1 != 0)
            feasibility++;
        if(c2 != 0)
feasibility++;  
    if(!Constraint1 <= 0)  
        feasibility++;  
    if(!Constraint2 <= 0)  
        feasibility++;  
}  
feasible = feasibility;  
if(feasibility == 0)  
{  
    funcvalue = 100*pow((x[1]-x[0]*x[0]), 2) + pow((x[0]-1), 2);  
    num_func_call++;  
}

//*************************************************************  
// Multi-modalFunction function  
//*************************************************************  

double MultimodalFunction(double x[], int &feasible, double &funcvalue, int n, int &num_func_call)  
{
    int feasibility = 0;  
    double sum =0, sum1 = 0,sum2 = 0;  
    double product = 1;  
    double Constraint1, Constraint2;  
    int i;  
    bool constraint = true;  

    for(i=0; i<n; i++)  
    {  
        sum += x[i];  
        product *= x[i];  
        if(!(x[i] >= 0 && x[i] <=10))  
            constraint = false;  
    }  
    Constraint1 = sum - 7.5*n;  
    Constraint2 = 0.75 - product;  
    if(constraint)  
    {  

if(Constraint1 <= 0 && Constraint2 <= 0)
    feasibility = 0;
else feasibility = 1;
}

else
{
    if(!constraint)
        feasibility++;
    if(!Constraint1 <= 0)
        feasibility++;
    if(!Constraint2 <= 0)
        feasibility++;
}

feasible = feasibility;

for(i=0; i<n ;i++)
    { 
        sum1 += pow(cos(x[i]), 4);
        product *= pow(cos(x[i]), 2);
        sum2 += (i+1)*pow(x[i], 2);
    }

if(feasibility ==0)
{
    funcvalue = (sum1 - 2*product)/ sqrt(sum2);
    if(funcvalue < 0)
        funcvalue = (-1)*funcvalue;
    num_func_call++;
}

//********************************************************************
// BenacerTao function
//********************************************************************

double BenacerTao(double x[], int &feasible, double &funcvalue, int n, int &num_func_call)
{
    int feasibility = 0;
    bool c1 = false, c2 = false;
    double func_value, Constraint;
    if(Constraint1 <= 0 && Constraint2 <= 0)
        feasibility = 0;
    else feasibility = 1;
}

else
{
    if(!constraint)
        feasibility++;
    if(!Constraint1 <= 0)
        feasibility++;
    if(!Constraint2 <= 0)
        feasibility++;
}

feasible = feasibility;

for(i=0; i<n ;i++)
    { 
        sum1 += pow(cos(x[i]), 4);
        product *= pow(cos(x[i]), 2);
        sum2 += (i+1)*pow(x[i], 2);
    }

if(feasibility ==0)
{
    funcvalue = (sum1 - 2*product)/ sqrt(sum2);
    if(funcvalue < 0)
        funcvalue = (-1)*funcvalue;
    num_func_call++;
}

//********************************************************************
// BenacerTao function
//********************************************************************

double BenacerTao(double x[], int &feasible, double &funcvalue, int n, int &num_func_call)
{
    int feasibility = 0;
    bool c1 = false, c2 = false;
    double func_value, Constraint;
if(x[0] >= 0)
c1 = true;
if(x[1] >= 0)
c2 = true;

Constraint = 2*x[0] - x[1] - 3;

if( c1 && c2 )
{
if(Constraint <= 0)
feasibility = 0;
else feasibility = 1;
}
else
{
if(!c1)
feasibility++;
if(!c2)
feasibility++;
if(!Constraint <= 0)
feasibility++;
}

feasible = feasibility;
if(feasibility == 0)
{
funcvalue = x[0] - x[1]/2 - 2*x[0]*x[0]/3 + x[1]*x[1]/2;
num_func_call++;
}

//********************************************************************
//
// RagsdellPhillips function
//********************************************************
void RagsdellPhillips(double x[], int &feasible, double & funcvalue, int n, int &num_func_call) {
    int i, feasibility = 0;
    double Constraint1, Constraint2, Constraint3, Constraint4, Constraint5, Constraint6;
    Constraint1 = 1.5211 - x[0]*x[1];
    Constraint2 = 16.8 - x[2]*x[3];
    Constraint3 = x[0]*x[3] - x[2];
    Constraint4 = 9.08 - x[2]*pow(x[3],2);
    Constraint5 = 0.09428*pow(x[3], 2)/pow(x[2], 3) + 0.02776*x[3] - 1;
    Constraint6 = 0.125 - x[0];

    Constraint = 2*x[0] - x[1] - 3;

    if( c1 && c2 )
    {
        if(Constraint <= 0)
            feasibility = 0;
        else feasibility = 1;
    }
    else
    {
        if(!c1)
            feasibility++;
        if(!c2)
            feasibility++;
        if(!Constraint <= 0)
            feasibility++;
    }
    feasible = feasibility;
    if(feasibility == 0)
    {
        funcvalue = x[0] - x[1]/2 - 2*x[0]*x[0]/3 + x[1]*x[1]/2;
        num_func_call++;
    }
if(x[0] > 0 && x[1] > 0 && x[2] > 0 && x[3] > 0)
{
    if(Constraint1 <= 0 && Constraint2 <= 0 && Constraint3 <= 0 && Constraint4 <= 0 && Constraint5 <= 0 && Constraint6 <= 0)
    {
        feasibility = 0;
    }
    else
    {
        feasibility = 1;
    }
}
else
{
    if(x[0] < 0)
    {
        feasibility ++;
    }
    if(x[1] < 0)
    {
        feasibility ++;
    }
    if(x[2] < 0)
    {
        feasibility ++;
    }
    if(x[3] < 0)
    {
        feasibility ++;
    }
    if(Constraint1 > 0)
    {
        feasibility ++;
    }
    if(Constraint2 > 0)
    {
        feasibility ++;
    }
    if(Constraint3 > 0)
    {
        feasibility ++;
    }
    if(Constraint4 > 0)
    {
        feasibility ++;
    }
    if(Constraint5 > 0)
    {
        feasibility ++;
    }
    if(Constraint6 > 0)
    {
        feasibility ++;
    }
}
feasible = feasibility;
if(feasibility == 0)
{
    funcvalue = 1.1047* pow(x[0], 2)*x[1] + 0.6735*x[2] + 0.04811*x[1]*x[2];
    num_func_call++;
}
}
void SpeedReducerProblem(double x[], int &feasible, double & funcvalue, int n, int &num_func_call)
{
    int i, feasibility=0;
    int c1, c2, c3, c4, c5, c6, c7;
    double f1, f2, f3, f4, f5, f6, f7;

    if(x[0] >= 2.6 && x[0] <= 3.6)
    {
        c1 = 0;
    }
    else
    {
        c1 = 1;
    }
    if(x[1] >= 0.7 && x[1] <= 0.8)
    {
        c2 = 0;
    }
    else
    {
        c2 = 1;
    }
    {
        c3 = 0;
    }
    else
    {
        c3 = 1;
    }
    {
        c4 = 0;
    }
    else
    {
        c4 = 1;
    }
    {
        c5 = 0;
    }
else
{
    c5 = 1;
}
if(x[5] >= 2.9 && x[5] <= 3.9)
{
    c6 = 0;
}
else
{
    c6 = 1;
}
if(x[6] >= 5.0 && x[6] <= 5.5)
{
    c7 = 0;
}
else
{
    c7 = 1;
}

if(c1 == 0 && c2 == 0 && c3 == 0 && c4 == 0 && c5 == 0 && c6 == 0 && c7 == 0)
    feasibility = 0;
else
{
    if(c1 == 1)
        feasibility++;
    if(c2 == 1)
        feasibility++;
    if(c3 == 1)
        feasibility++;
    if(c4 == 1)
        feasibility++;
    if(c5 == 1)
        feasibility++;
    if(c6 == 1)
        feasibility++;
    if(c7 == 1)
        feasibility++;
}

feasible = feasibility;

f1 = 0.7845*x[0]*pow(x[1], 2);
f2 = 3.3333*pow(x[1], 2) + 14.9334*x[2] - 43.0934;
f3 = pow(x[1], 2) + pow(x[6], 2);
f4 = pow(x[5], 3) + pow(x[6], 3);
f5 = 0.7854*(x[3]*pow(x[5], 2) + x[4]*pow(x[6], 2));
f6 = -1.508*x[0]*f3;
\begin{verbatim}
f7 = 7.477*x[0]*f4;
f1 = f1*f2;

if(feasible == 0)
{
    funcvalue = f1 + f6 + f7 + f5;
    num_func_call++;
}

//********************************************************************
// // RosenSuzuki Function // //************************************************************************
void RosenSuzukiFunction(double x[], int &feasible, double & funcvalue, int n, int &num_func_call)
{
    int feasibility = 0;
    double Constraint1, Constraint2, Constraint3;


    if(Constraint1 <= 0 && Constraint2 <= 0 && Constraint3 <= 0)
    {
        feasibility = 0;
    }
    else
    {
        if(Constraint1 > 0)
            feasibility ++;
        if(Constraint2 > 0)
            feasibility ++;
        if(Constraint3 > 0)
            feasibility ++;
    }

    feasible = feasibility;
    if(feasibility == 0)
    {
        num_func_call++;
    }
\end{verbatim}
void PressureVessel(double x[], int &feasible, double & funcvalue, int n, int &num_func_call)
{
    int feasibility;

    double Constraint1, Constraint2, Constraint3, Constraint4;

    Constraint1 = 0.0193*x[2] - x[0];
    Constraint2 = 0.00954*x[2] - x[1];
    Constraint3 = 1296000 -pi*pow(x[2], 2)*x[3] - (4/3)*pi*pow(x[2], 3);
    Constraint4 = x[3] - 240;

    if(Constraint1 <= 0 && Constraint2 <= 0 && Constraint3 <= 0 && Constraint4 <= 0)
    {
        feasibility = 0;
    }
    else
    {
        if(Constraint1 > 0)
            feasibility++;
        if(Constraint2 > 0)
            feasibility++;
        if(Constraint3 > 0)
            feasibility++;
        if(Constraint4 > 0)
            feasibility++;
    }

    feasible = feasibility;
    if(feasibility == 0)
    {
        funcvalue = 0.6224*x[0]*x[2]*x[3] + 1.7781*x[1]*pow(x[2], 2) +
                    3.1661*pow(x[0], 2)*x[3] + 19.84*pow(x[0], 2)*x[2];
        num_func_call++;
    }
}
void HimmelblauProblem(double x[], int &feasible, double &funcvalue, int n, int &num_func_call)
{
    int feasibility;
    double Constraint1, Constraint2, Constraint3;
    int c1, c2, c3, c4, c5, c6, c7, c8;

    Constraint1 = 85.334407 + 0.0056858*x[1]*x[4] + 0.00026*x[0]*x[3] -
                    0.0022053*x[2]*x[4];
    Constraint2 = 80.51249 + 0.0071317*x[1]*x[4] + 0.0029955*x[1]*x[0] +
                    0.0021813*pow(x[2], 2);
    Constraint3 = 9.300961 + 0.0047026*x[2]*x[4] + 0.0012547*x[0]*x[2] +
                    0.0019085*x[2]*x[3];

    if(x[0] >= 78 && x[0] <= 102)
        c1 = 0;
    else c1 = 1;
    if(x[1] >= 33 && x[1] <= 45)
        c2 = 0;
    else c2 = 1;
        c3 = 0;
    else c3 = 1;
        c4 = 0;
    else c4 = 1;
        c5 = 0;
    else c5 = 1;
    if(Constraint1 >= 0 && Constraint1 <= 92)
        c6 = 0;
    else c6 = 1;
    if(Constraint2 >= 90 && Constraint2 <= 110)
        c7 = 0;
    else c7 = 1;
    if(Constraint3 >= 20 && Constraint3 <= 25)
        c8 = 0;
    else c8 = 1;

    if(c1 == 0 && c2 == 0 && c3 == 0 && c4 == 0 && c5 == 0 && c6 == 0 && c7
      == 0 && c8 == 0)
    {
        feasibility = 0;
    }
}
else
{
  if(c1 == 1)
    feasibility++;  
  if(c2 == 1)
    feasibility++;  
  if(c3 == 1)
    feasibility++;  
  if(c4 == 1)
    feasibility++;  
  if(c5 == 1)
    feasibility++;  
  if(c6 == 1)
    feasibility++;  
  if(c7 == 1)
    feasibility++;  
  if(c8 == 1)
    feasibility++;  
}

feasible = feasibility;

if(feasible == 0)
{
  funcvalue = 5.3578547*pow(x[2], 2) + 0.8356892*x[0]*x[4] + 37.29329*x[0] - 40792.141;
  num_func_call ++;
}

//********************************************************************
//
// BenchmarkProblem4 function
//
//********************************************************************

void BenchmarkProblem4(double x[], int &feasible, double & funcvalue, int n, int &num_func_call)
{
  int feasibility, i;
  double sum1 = 0.0, sum2 = 0.0;
  double Constraint1, Constraint2, Constraint3, Constraint4, Constraint5,
Constraint6, Constraint7, Constraint8, Constraint9;

  Constraint1 = 2*x[0] + 2*x[1] + x[9] + x[10];
  Constraint4 = x[9] - 8*x[0];
Constraint5 = x[10] - 8*x[1];
Constraint6 = x[11] - 8*x[2];

for(i=0; i<4; i++)
    sum1 += pow(x[i], 2);
for(i=4; i<13; i++)
    sum2 += pow(x[i], 2);

if(Constraint1 <= 10 && Constraint2 <= 10 && Constraint3 <= 10 &&
Constraint4 <= 0 && Constraint5 <= 0 && Constraint6 <= 0 && Constraint7 <= 0 &&
Constraint8 <= 0 && Constraint9 <= 0)
{
    feasibility = 0;
}
else
{
    if(Constraint1 > 10)
        feasibility++;
    if(Constraint2 > 10)
        feasibility++;
    if(Constraint3 > 10)
        feasibility++;
    if(Constraint4 > 0)
        feasibility++;
    if(Constraint5 > 0)
        feasibility++;
    if(Constraint6 > 0)
        feasibility++;
    if(Constraint7 > 0)
        feasibility++;
    if(Constraint8 > 0)
        feasibility++;
    if(Constraint9 > 0)
        feasibility++;
}
feasible = feasibility;

if(feasible == 0)
{
    num_func_call++;
}
void Problem4(double x[], int &feasible, double & funcvalue, int n, int &num_func_call)
{
    int feasibility;
    double Constraint1, Constraint2, Constraint3, Constraint4, Constraint5, Constraint6;
    Constraint1 = 1 - 0.0025*(x[3] + x[5]);
    Constraint3 = 1 - 0.01*(x[7] - x[4]);
    Constraint4 = x[0]*x[5] - 833.33252*x[3] - 100*x[0] + 83333.333;

    if(Constraint1 >= 0 && Constraint2 >= 0 && Constraint3 >= 0 && Constraint4 >= 0 && Constraint5 >= 0 && Constraint6 >= 0)
    {
        feasibility = 0;
    }
    else
    {
        if(Constraint1 < 0)
            feasibility++;
        if(Constraint2 < 0)
            feasibility++;
        if(Constraint3 < 0)
            feasibility++;
        if(Constraint4 < 0)
            feasibility++;
        if(Constraint5 < 0)
            feasibility++;
        if(Constraint6 < 0)
            feasibility++;
    }
    feasible = feasibility;

    if(feasible == 0)
    {
        funcvalue = x[0] + x[1] + x[2];
        num_func_call++;
    }
}