Computer Assisted Mechanics and Engineering Sciences, 11: 155–165, 2004.

Optimal design of engineering systems using MPI-enabled genetic algorithm

S. D. Rajan[1], D. T. Nguyen[2], M. D. Deshpande[3], L. Harrell[4]
[1] *Professor, Department of Civil Engineering, Arizona State University, Tempe, AZ 85287, USA,*
[2] *Professor, Department of Civil and Environmental Engineering, Old Dominion University, Norfolk, VA 23539, USA,*
[3] *Electromagnetic Research Branch, NASA Langley Research Center, Hampton, VA 23681, USA,*
[4] *Assistant Professor, Department of Civil and Environmental Engineering, Old Dominion University, Norfolk, VA 23539, USA*

(Received December 12, 2002)

The focus of this paper is on the development and implementation of a genetic algorithm (GA)–based software system using message passing interface (MPI) protocol and library. A customized form of simple GA used in previous research [1–4] is parallelized. This MPI-enabled version is used to find the solution to finite element based design optimization problems. Results show that an almost linear speedup is obtained on homogenous hardware cluster and, with proper reworking of the software, on heterogeneous hardware cluster.

Keywords: Parallel processing, genetic algorithm, MPI, structural optimization

1. Introduction

Genetic algorithms (GA) have evolved over the last three decades to be recognized as a very powerful tool in obtaining solutions to non-engineering and engineering design optimization problems. The simple GA while powerful, is perhaps too general to be efficient and robust for structural design problems. First, function (or, fitness) evaluations are computationally expensive since they typically involve finite element analysis. Second, the (feasible) design space is at times disjointed with multiple local minima. Third, the design space can be a function of boolean, discrete and continuous design variables. The use of GA to find the optimal solution(s) of engineering design problems is still an open research area. Experience with GA has indicated that more often than not, tuning the GA strategy and parameters can lead to more efficient solution process for a class of problems.

One of the most interesting aspects of GA is the explosive growth in the number of strategies explored by researchers in a multitude of disciplines. Below, we present a fraction of a number of GA variations used in advancing the applicability of GAs especially when parallel computing is available.

Goodman et al. [5] present an approach to optimal design of elastic flywheels using an Injection Island Genetic Algorithm (iiGA). An iiGA in combination with a finite element code is used to search for shape variations to optimize the Specific Energy Density (SED) of elastic flywheels. SED is defined as the amount of rotational energy stored per unit mass. iiGAs seek solutions simultaneously at different levels of refinement of the problem representation (and correspondingly different definitions of the fitness function) in separate sub-populations (islands). Solutions are sought first at low levels of refinement with an axisymmetric plane stress finite element code for high speed exploration of the coarse design space. Next, individuals are injected into populations with a higher level of resolution that uses an axisymmetric three dimensional finite element model to "fine-tune" the flywheel designs. In true multi-objective optimization, various "sub-fitness" functions can be defined that represent "good" aspects of the overall fitness function. Solutions can be sought for these various "sub-fitness" functions on different nodes and injected into a node that evaluates the overall fitness. Allowing subpopulations to explore different regions of the fitness space simultaneously allows relatively robust and efficient exploration in problems for which fitness evaluations are costly.

Miki et al. [6] present an approach where the whole population is divided into several subpopulations which are called islands. A migration scheme that moves some individuals in one island to another island is adopted to create new population mixes. In the proposed approach, different values of the mutation rate and the crossover rate are assigned to different islands thereby creating different GA environments in each of these islands. The optimization problem that is solved is the minimization of the volume of truss structures under tensile, buckling, and displacement constraints.

Sarma and Adeli [7] use a bilevel strategy in finding the solution to the design problem using GAs. First, parallel fuzzy GAs are used to obtain a continuous-variable minimum weight design. In this stage, the objective function and the constrains are considered to be fuzz and a genetic search is performed with a preemptive constraint-violation strategy. Small constraint violations are allowed. This solution is then used as a preliminary startup design for the subsequent fuzzy discrete multicriteria cost optimization. Both OpenMP directive and MPI calls are used in a shared memory data parallel computing and message passing distributed computing to take advantage of the best of both approaches.

Scott et al. [8] present a hardware-based GA solution methodology. Speedups of 1-3 orders of magnitude have been observed when frequently used software routines were implemented in hardware by way of reprogrammable field-programmable gate arrays (FPGAs). Reprogrammability is essential in a general-purpose GA engine because certain GA modules require changeability (e.g. the function to be optimized by the GA). Thus a hardware-based GA is both feasible and desirable. A fully functional hardware-based genetic algorithm (the HGA) is presented here as a proof-of-concept system. It was designed using VHDL to allow for easy scalability. It is designed to act as a coprocessor with the CPU of a PC. The user programs the FPGAs which implement the function to be optimized. Other GA parameters may also be specified by the user. Simulation results and performance analyses of the HGA are presented. A prototype HGA is described and compared to a similar GA implemented in software. In the simple tests, the prototype took about 6% as many clock cycles to run as the software-based GA. Further suggested improvements could realistically make the HGA 2–3 orders of magnitude faster than the software-based GA.

We have two major focus or objectives in this paper. First, the proposed improvements to the simple GA are discussed. These improvements are aimed at improving the reliability and efficiency of the overall process. Second, the parallel implementation is such that load-balancing issue is tackled so that the overall methodology works efficiently on both homogenous and heterogeneous computer clusters.

In the discussions that follow in the paper, the following definitions are used. When explaining methodologies in the general sense, we define efficient, reliable, accurate and robust methods as follows.

Efficient: AA methodology is defined as being efficient if it finds an acceptable solution with minimal computational effort.

Reliable: A methodology is defined as being reliable if it finds an acceptable solution regardless of the problem nuances or the starting point used.

Accurate: A methodology is defined as being accurate if it finds the best possible solution to a problem.

Robust: A methodology that is generally efficient, reliable and accurate.

In addition, we define a homogenous computing cluster as follows.

Homogenous Cluster: A homogenous cluster is defined as one having identical computers (or nodes) connected by a switch (a heterogeneous cluster is one where the nodes are not identical).

2. Genetic algorithm

The design problem can be stated as follows.

Find $\mathbf{x} = \lfloor {}^b x_1, \ldots, {}^b x_{n_b}; {}^i x_1, \ldots, {}^i x_{n_d}; {}^s x_1, \ldots, {}^s x_{n_s} \rfloor$ to minimize $f(\mathbf{x})$ subject to $g_i(\mathbf{x}) \le 0$ $i = 1, \ldots, n_i$

$$h_j(\mathbf{x}) = 0 \qquad j = 1, \ldots, n_e, \tag{1}$$

$$^b x_p \in \{ 0, 1 \} \qquad p = 1, \ldots, n_b, \tag{2}$$

$$^i x_q \in \{ x_q^1, x_q^2, \ldots\ldots, x_q^{nq} \} \qquad q = 1, \ldots, n_d, \tag{3}$$

$$^s x_r^L \le {}^s x_r \le {}^s x_r^U \qquad r = 1, \ldots, n_s, \tag{4}$$

where $\mathbf{x}$ is the design variable vector, $f(\mathbf{x})$ is the objective function, n_i is the number of inequality constraints, n_e is the number of equality constraints, n_b is the number of boolean design variables, n_d is the number of discrete design variables selected from a list of nq values, and n_s is the number of continuous design variables. The genetic algorithm used in this research has evolved and refined over time. In this section we discuss some of the improvements that have been made to a simple GA in order to improve its overall performance. Further details can be found in prior publications [1–4].

(1) *Adaptive penalty function*: GAs were developed to solve unconstrained optimization problems. However, engineering design problems are usually constrained. They are solved by transforming the problem to an unconstrained problem. The transformation is not unique and one possibility is to use the following strategy.

$$\text{minimize}: \quad f(\mathbf{x}) + \sum_i c_i \cdot \max(0,\ g_i) + \sum_j c_j \cdot |h_j| \tag{5}$$

where c_i and c_j are penalty parameters used with inequality and equality constraints. Determining the appropriate penalty weights c_i and c_j is always problematic. We use an algorithm here where the penalty weight is computed automatically based on the traits of the current population and adjusted in an adaptive manner.

(2) *Improving crossover operators using the Association String*: As discussed by some researchers, the one-point crossover is preferred for continuous domains, and the uniform crossover for discrete domains. However, schema representation still plays a pivotal role in the efficiency of the GA. If one uses a one-point crossover then it is obvious that the ordering of the design variables is an important issue. Since the characteristic of one-point crossover is that the shorter schema has a better chance to survive, if two variables that have less of an interdependency are placed adjacent to each other, or two variables with a strong relationship are placed far away from each other, the crossover operation will make it more difficult for the GA to search the design space efficiently. To implement this strategy, we introduce an additional string called the *Association String*. Results show that the Association String improves the robustness of the solution process.

(3) *Mating Pool Selection*: The selection scheme (for generating the mating pool) together with the penalty function dictate the probability of survival of each string. While it is very important to preserve the diversity in each generation, researchers have also found that sometimes it may be profitable to bias certain schema. However, results from most of the selection rules, like roulette wheel, depend heavily on the mapping of fitness function. In this paper, the tournament selection is used. There are at least two reasons for this choice. First, tournament selection increases the probability of survival of better strings. Second, only the relative fitness values are relevant when comparing two strings. In other words, the selection depends on individual fitness rather than ratio of fitness values. This is attractive since in this research, the fitness value contains the penalty term and does not represent the true objective function.

(4) *Elitist Approach*: Research has shown that the GA with the incorporation of the elitist approach can be more reliable and efficient than the ones without. This approach is used in the current research.

(5) *Population Size and Stopping Criteria*: Generally speaking, the initial population should contain uniformly distributed alleles. By this it is meant that, if possible, no chromosome pattern should be missed. Each chromosome is represented by n bits with each bit being either 1 or 0. If the distribution of 1's (or 0's) in each bit location is to be uniform, the initial population size should be at least n. During the evolution, it is expected that that the chromosome converges to some special pattern with the 0–1 choice decided for n locations. Assume that the choice of each bit is independent of all the other bits. Since the population size is n in each generation, after every generation from the statistical viewpoint we can expect to learn about at least one bit. Ideally then after n generations, one can expect to learn about all the n bits forming the chromosome. However, since each bit is not independent of the others, more than n generations are perhaps necessary to obtain a good solution. This suggests that the population size and the number of generations should be *at least* n. Numerical experience in our previous work suggests that using population and generation size of $2n$ leads to acceptable results efficiently.

3. Parallel GA

The overall algorithm used in a GA-based design optimization problem is quite simple. The overall flow is shown in Fig. 1. For engineering design problems, from a computational viewpoint, the fitness evaluation is the most expensive step. Hence, it would be prudent to parallelize the fitness evaluation step.

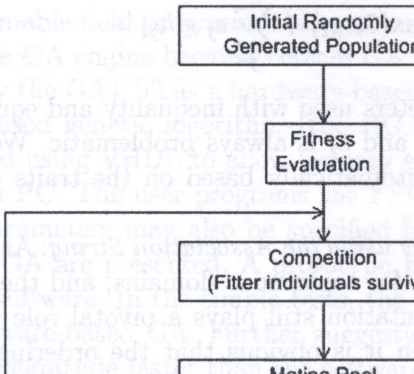
Initial Randomly Generated Population
Fitness Evaluation
Competition (Fitter individuals survive)
Mating Pool (Reproduction phase)
Crossover and Mutation (Exchange of information)
Offsprings (New generation)
Fitness Evaluation
No
Stop?

Fig. 1. Flow in a Simple Genetic Algorithm (SGA)

There are two algorithms that can be used in evaluating the fitness function in parallel. The first version is presented next. We will assume that the number of available processes, n_p is less than the population size, n_{pop}.

Send All-Then-Receive (SATR) Version

Master Process

```
1. Set next available process as process $j = 0$.
2. Loop through all members of the population, $i = 1, 2, ..., n_{\text{pop}}$.
3. Generate the vector of design variables, $\mathbf{x}$.
4. Pass this vector to process $j$.
5. Increment $j$. If $j = n_p$, set $j = 0$.
6. End loop.
7. Set next process as process $j = 0$.
8. Loop through all members of the population, $i$.
9. Receive the objective function value and the maximum constraint violation from process $j$.
10. Increment $j$. If $j = n_p$, set $j = 0$.
11. End loop.
```

Slave Process

```
1. Set next available process as processor $j = 0$.
2. Loop through all members of the population, $i = 1, 2, ..., n_{\text{pop}}$.
3. If $j$ is equal to the slave process number, receive the vector of design variables, compute and
   send the objective function value and the maximum constraint violation.
4. Increment $j$. If $j = n_p$, set $j = 0$.
5. End loop.
```

The problem with the above algorithm is that (a) for the master process, 'receives' do not start until all the 'sends' are completed, (b) for the slave process, 'sends' do not start until all the 'receives' are completed, and (c) load imbalance will take place in a heterogeneous computing environment. Under this scenario, a buffer overflow is likely to occur. Here is a typical error message that one is likely to encounter.

MPI/Pro: [4] : Too many unexpected messages! Internal TCP buffer space exhausted

A modified version is presented next where send and receive take place one after the other literally on-demand.

Load-Balanced (LB) Version

Master Process

```
1. Set number of messages sent, $n_{\text{sent}} = 0$.
2. Loop through $j = 0, ..., \min(n_{\text{pop}}, n_p)$.
3. Generate the vector of design variables, $\mathbf{x}$. Pass this vector to process $j$. Increment $n_{\text{sent}}$.
4. End loop.
5. Loop through all members of the population, $i = 1, 2, ..., n_{\text{pop}}$.
6. Receive the objective function value and the maximum constraint violation from process $j$.
7. If $n_{\text{sent}} < n_{\text{pop}}$, generate the vector of design variables, $\mathbf{x}$ for member $i$. Pass this vector to
   process $j$. Increment $n_{\text{sent}}$. Else send message to process $j$ that this is the last message.
8. End loop.
```

Slave Process (Valid only for process $j < n_{\text{pop}}$)

1. Loop until last message received.
2. If not last message, receive the vector of design variables, compute and send the objective function value and the maximum constraint violation.
3. End loop.

It should be noted that in both the versions, only the master process executes the GA. The implication is that the communication time is held to a minimum. Assuming that one integer word is 4 bytes, one double precision word is 8 bytes and the objective function and maximum constraint violation are designated as double precision, we can compute the total number of send and receive bytes as follows for every generation.

$$n_{\text{send}} = n_{\text{receive}} = 4n_{\text{pop}}\left(n_b + n_d + 2n_s\right) + 8\left(2n_{\text{pop}}\right) \tag{6}$$

With both these approaches, it is desirable to use MPI_Barrier after the entire population is evaluated so that the execution remains synchronized on all the processes.

4. Numerical examples

The focus of the current research is to develop and test a parallel, MPI-enabled GA for engineering problems. Hence, only a sizing optimal design problem is solved using an academic problem that has the desired characteristics (number of degrees-of-freedom and design variables) to test the parallel implementation. The results and conclusions, as we will discuss later, can be easily extended to other types of structural design problems.

The sizing optimization problem is as follows.

Find

$$\mathbf{x}_{k\times 1}. \tag{7}$$

To

$$\min f(\mathbf{x}) = \sum_{i=1}^{n} A_i L_i \rho_i \tag{8}$$

subject to

$$g_i \equiv \sigma_i \leq \sigma_a, \qquad i = 1, 2, ..., n, \tag{9}$$

$$x_j^L \leq x_j \leq x_j^U, \qquad j = 1, 2, ..., k, \tag{10}$$

where $\mathbf{x}_{k\times 1}$ represents the cross-sectional areas of the truss members (design variables), $f(\mathbf{x})$ represents the mass of the truss (objective function) and g_i the stress constraints. To evaluate the fitness evaluation one must compute the objective function and all the constraints. Without resorting to any approximation technique, a full finite element analysis is required to evaluate the fitness as shown in Eq. (2).

Hardware: The numerical examples were generated on two different clusters. The first is labeled as High-Cost Cluster and is made up of more expensive computers and a high-performance switch. The second is labeled as a low-cost cluster and commodity computers and switch are used.

High-Cost Homogenous Cluster Information: (a) Number of machines in the cluster = 7 (b) Typical machine: Intel P4 1.7 GHz Dual Xeon, 512 MB RDRAM, Ultra 7200 rpm IDE Drive, Intel PRO/1000 T NIC, (c) Windows 2000 (SP 2), MPI-Softtech 1.6.3 [9], Cisco Catalyst 3550-12T switch.

Low-Cost Heterogeneous Cluster Information: (a) Number of machines in the cluster = 2 (b) Computer 1: Intel P3-866 MHz, 768 MB RDRAM, Ultra 7200 rpm IDE Drive, 3COM 3C920 NIC. Computer 2: AMD 1.2 GHz Athlon, 512 MB SDRAM, Ultra 7200 rpm IDE Drive, 3COM 3C920 NIC, (c) Windows 2000 (SP 2), MPI-Softtech 1.6.3, Linksys BEFSR41 10/100 Router.

Test Problems: The structural system that is designed is shown in Fig. 2. The planar truss is described in terms of two parameters – the number of bays and the number of storeys. The truss members are grouped into three groups per storey – horizontal members, vertical members and diagonal members. Hence, the number of design variables is equal to three times the number of storeys. The specific values used in the following examples are as follows.

$\rho = 0.00881448$ lbm/in^3
$\sigma_a = 10000$ psi
$x_j^L = 0.1\,\text{in}^2$ and $x_j^U = 20\,\text{in}^2$ and precision is taken as 0.1 in^2,
Bay width = 240 in
Storey height = 120 in
Applied load, P=10000 lb

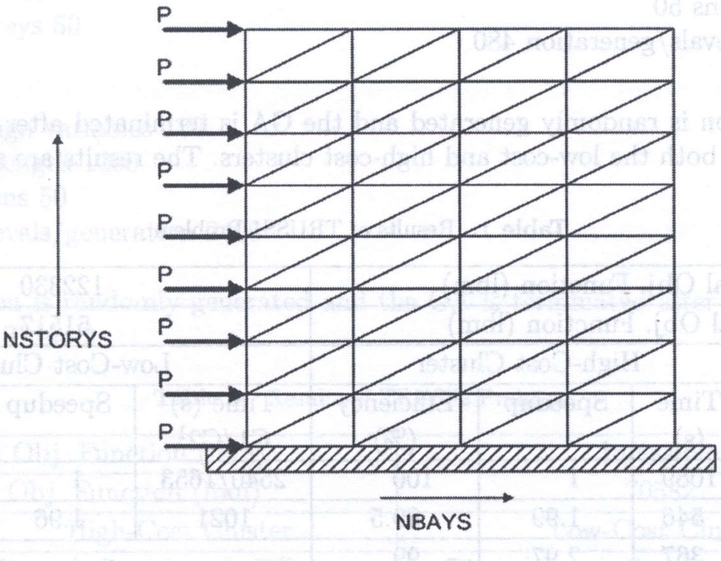

Fig. 2. Layout of the planar truss

Two test problems are solved and are identified as TRUSS1 and TRUSS2. TRUSS1 is a structurally larger problems meaning that one complete finite element analysis takes more time compared to the other model. However, the number of design variables is smaller. TRUSS2, on the other hand, is different – smaller structural model but has much larger number of design variables. The problem details and results are presented next. All timing information is in terms of wall clock (or elapsed) time.

For a homogenous system, we compute the speedup and efficiency as follows.

$$\text{Speedup obtained for } n \text{ processes} = \frac{\text{Time for one process}}{\text{Time for } n \text{ processes}}, \tag{11}$$

$$\text{Efficiency for } n \text{ processes} = \frac{\text{Speedup obtained}}{n} \times 100. \tag{12}$$

For a heterogeneous system, we compute the speedup as follows. Let the relative speed of the n_p processes be denoted as $s_1, s_2, ..., 1, ..., s_{n_p}$, the time taken for each single process run be denoted as $t_1, t_2, ..., t_i, ..., t_{n_p}$, and t is the time taken when n processes are used. Then

$$s_j = \frac{\max(t_1, t_2, ..., t_{n_p})}{t_j}. \tag{13}$$

$$\text{Speedup obtained for } n \text{ processes} = \frac{\frac{1}{\sum_{j=1}^{n_p} s_j}\left(\sum_{i=1}^{n_p} s_i t_i\right)}{t}. \tag{14}$$

Test Problem 1 (ID: TRUSS1): The problem-specific data are as follows.

Number of bays 300
Number of storeys 10
Nodes 3311
Elements 9010
Number of design variables 30
Chromosome Length 240
# of Generations 50
# of function evals/generation 480

The initial population is randomly generated and the GA is terminated after 50 generations. The problem is solved in both the low-cost and high-cost clusters. The results are shown in Table 1.

Table 1. Results of TRUSS1 Problem

Initial Obj. Function (lbm)				122330		
Final Obj. Function (lbm)				51517		
	High-Cost Cluster			Low-Cost Cluster		
# of Processes	Time (s)	Speedup	Efficiency (%)	Time (s) C1/C2[1]	Speedup	Efficiency (%)
1	1089	1	100	2540/1653	1	100
2	546	1.99	99.5	1021	1.96	98
3	367	2.97	99			
4	277	3.93	98.3			
5	224	4.86	97.2			
6	191	5.70	95			

Remarks: The timing values show that the GA-runs scale almost linearly in the high-cost (homogenous) cluster. A relatively modest 245760 bytes are sent and received every generation. The low-cost heterogeneous cluster shows an almost linear speedup indicating that the LB version of distributing the fitness evaluations does a reasonable job – the number of fitness evaluation computed by C1 and C2, on an average, is 170 and 310 respectively. The design history is shown in Fig. 3. In a separate run to assess the quality of the solution obtained, the objective function was found to reduce to 34726 lbm after 100 generations.

[1]C1/C2: Computer 1 and Computer 2

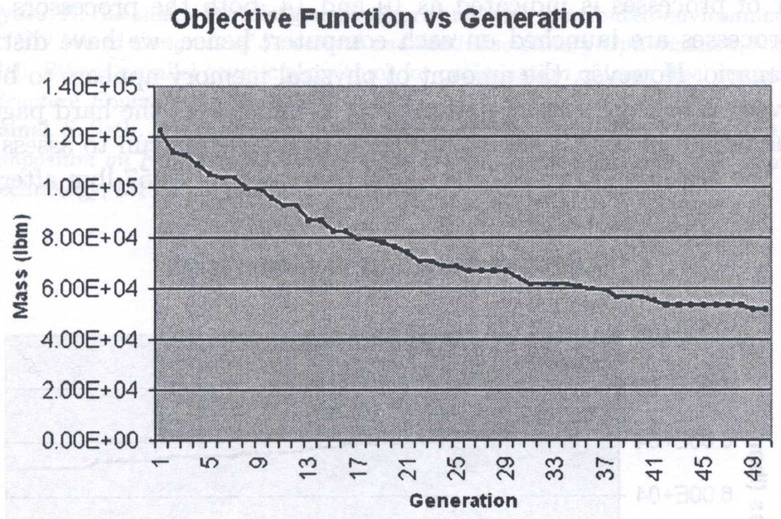

Fig. 3. Design history showing objective function versus generation (TRUSS1)

Test Problem 2 (ID: TRUSS2): The problem-specific data are as follows.

Number of bays 30
Number of storeys 50
Nodes 1581
Elements 4550
Number of design variables 150
Chromosome Length 1200
# of Generations 50
# of function evals/generation 2400

The initial population is randomly generated and the GA is terminated after 50 generations. The results are shown in Table 2.

Table 2. Results of TRUSS2 Problem

Initial Obj. Function (lbm)				Infeasible		
Final Obj. Function (lbm)				70582		
	High-Cost Cluster			Low-Cost Cluster		
# of Processes	Time (s)	Speedup	Efficiency (%)	Time (s) C1/C2	Speedup	Efficiency (%)
1	6762	1	100	18491/11930	1	100
2	3366	2.0	100	74742	1.94	97
3	2269	2.98	93.3			
4	1709	3.96	99			
5	1382	4.89	97.8			
6	1155	5.85	97.5			
10	716	9.44	94.4			
14	520	13.0	92.9			

Remarks: Once gain we see an almost linear speedup with the high-cost cluster. Network traffic is much more in this example. A total of 5836800 bytes are sent and received every generation.

When the number of processes is indicated as 10 and 14, both the processors on each computer were used. Two processes are launched on each computer; hence, we have distributed as well as shared memory scenario. However, the amount of physical memory appears to be adequate for the problem being solved. When the system performance is monitored, the hard page faults are shown to be minimal. The design history is shown in Fig. 4. In a separate run to assess the quality of the solution obtained, the objective function was found to reduce to 64457 lbm after 100 generations.

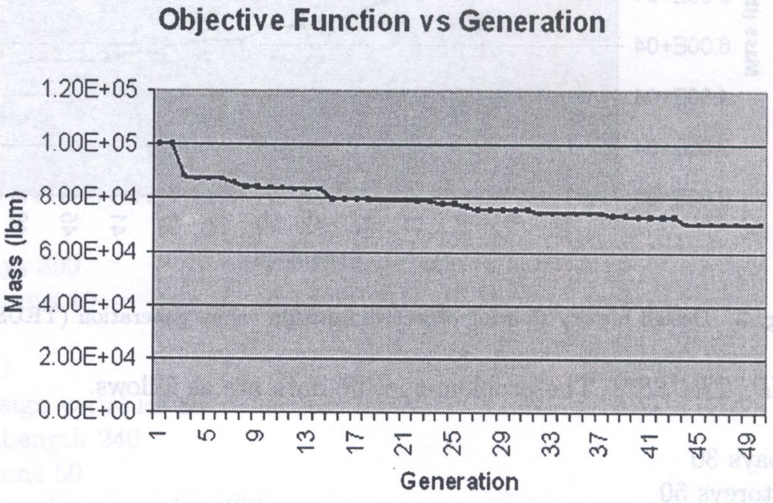

Fig. 4. Design history showing objective function versus generation (TRUSS2)

5. Concluding Remarks

The development and implementation of an MPI-enabled GA is discussed. Due to the very nature of genetic algorithms, linear speedup is possible with minimal effort – a good example of an embarrassingly parallel problem. As we can see from Eq. 6, communication time is a function of the number of design variables and the population size. Strategies can be developed to reduce the communication time, especially in a homogenous environment, by having the master process broadcast the fitness values of the entire population to all the slave processes. With this approach, the design variables do not have to be sent to all the slave processes. There are other approaches that can be taken to improve the overall performance of the GA. For example, the concept of using DGA with subpopulations and migration between these population islands is suitable for a parallel computing environment. These and other ideas are currently being investigated.

References

[1] S-Y. Chen, J. Situ, B. Mobasher, S. D. Rajan, Use of genetic algorithms for the automated design of residential steel roof trusses. *ASCE Press*, 43–54, 1997.

[2] S-Y. Chen, S. D. Rajan, Improving the efficiency of genetic algorithms for frame designs. *Engineering Optimization*, 30, 281–307, 1998.

[3] S. D. Rajan, B. Mobasher, S-Y.Chen, C. Young. Cost-based design of residential steel roof systems: a case study. *Struct. Eng. Mech.*, 8–2, 165–180, 1999.

[4] S-Y. Chen, S. D.Rajan. A robust genetic algorithm for structural optimization. *Struct. Eng. Mech.*, 10–4, 313–336, 2000.

[5] D. Eby, R. C. Averill, B. Gelfand, W. F. Punch, III, O. Mathews, E. D. Goodman. An injection island GA for flywheel design optimization. *Proc. EUFIT '97, – 5th European Congress on Intelligent Techniques and Soft Computing*, 1997.

[6] M. Miki, T. Hiroyasu, K. Hatanaka. Parallel genetic algorithms with distributed-environment multiple population scheme. 3rd *WCSMO World Congress of Structural and Multidisciplinary Optimization*, Niagara Falls, NY, 1999.

[7] K. Sarma, H. Adeli. Bilevel parallel genetic algorithms for optimization of large steel structures. *Computer-Aided Civil and Infrastructure Engineering*, 16, 295–304, 2001.

[8] S. D. Scott, A. Samal, S. Seth. HGA: a hardware–based genetic algorithm. *Proc. of the 1995 ACM/SIGDA Third International Symposium on Field-Programmable Gate Arrays*, 53–59, Monterey, CA, February 1995.

[9] MPI Software Technology, MPI-Pro Version 1.6.3, 2002.