MASTER THESIS

Development Of A Framework For Structural Optimization Using Parallel Computers

by

© Anh Quan, Nguyen
Date: 21 August, 2011

Advisor: Dr. Karlheinz Lehner
Supervisor: Prof. Dr.-Ing. Markus König

Chair of Computing in Engineering
Faculty of Civil and Environmental Engineering

Ruhr University of Bochum
The aim of structural optimization is to find a variant of a structural system that is some well-defined sense the “best” solution regarding to costs or weight while satisfying all relevant engineering constraints, such as maximum allowable stress in each member. Moreover, the optimization of a system implies that many responses must be determined, so the optimization of large structural system can only be done successfully by using massive parallel computer hardware and parallel programming paradigms. Using the minife software, available as Java library software package to define structural systems, a prototype software system is developed to show how parallel programming paradigms, such as the master-worker programming model, can form the basis of a framework for structural optimization based on parallel computers.

Keywords: structural optimization, engineering constraints, massive parallel computer hardware, parallel programming paradigms, minife, prototype, master-worker.
ACKNOWLEDGEMENTS

The research for this master thesis has been carried out under the supervision of Dr. Karlheinz Lehner and Prof. Dr.-Ing. Markus König, Chair of Computing in Engineering, Faculty of Civil and Environmental Engineering, Ruhr University of Bochum.

This thesis is a result of the guidance and support of several people who have to be sincerely acknowledged.

I would like to take this opportunity to thank Prof. Dr. Ing. Markus König, and Dr. Karlheinz Lehner for allowing me to carry out the master thesis at Department of Computing in Engineering. I would like to express my sincere gratitude to my thesis advisor Dr. Karlheinz Lehner for his guidance, support and constant encouragement that have made this quest a success. I also thank him for providing me with valuable suggestions and comments during my work. His suggestions helped me improve my problem solving capabilities, which encouraged me to never give up on any problem until it is rightfully solved. This thesis would have not become possible without his support.

I also thank my friends for their help, support, and presence in my life during my study in Ruhr University of Bochum.

Finally, I would like to thank my family and my love to all the sacrifice they have done for me in order to succeed throughout my life and for their unconditional love and encouragement.
Table of Contents

Chapter I ................................................................................................................................. 11
Parallel Computing .................................................................................................................. 11
  1. Introduction ....................................................................................................................... 11
    1.1 Background ..................................................................................................................... 11
    1.2 Amdahl's law and Gustafson’s law .............................................................................. 12
    1.3 Bernstein conditions ................................................................................................. 13

  2. Parallel Programming ...................................................................................................... 15
    2.1 Introduction .................................................................................................................... 15
    2.2 OpenMP ........................................................................................................................ 17
    2.3 MPI .................................................................................................................................. 19

  3. MPJ ................................................................................................................................... 20
    3.1 Introduction ..................................................................................................................... 20
    3.2 MPJ Design .................................................................................................................... 20
    3.3 MPJ implementation ...................................................................................................... 22

Chapter II .............................................................................................................................. 24
Framework with Master-Worker Paradigm .............................................................................. 24
  1. Master-Worker Paradigm Introduction ........................................................................... 24

  2. Master-Worker Architecture ............................................................................................ 26
    2.1 Master Module ............................................................................................................... 26
    2.2 Worker Module ............................................................................................................. 26

  3. Implementation of Framework with Master-Worker Paradigm .................................... 27
    3.1 Main Program Flow ....................................................................................................... 27
    3.2 Master-Worker Flow ..................................................................................................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Plane Truss</td>
<td>78</td>
</tr>
<tr>
<td>3. Performance</td>
<td>79</td>
</tr>
<tr>
<td>Chapter VI</td>
<td>85</td>
</tr>
<tr>
<td>Conclusion and Future Work</td>
<td>85</td>
</tr>
<tr>
<td>1. Conclusions</td>
<td>85</td>
</tr>
<tr>
<td>2. Future Work</td>
<td>85</td>
</tr>
<tr>
<td>Appendix A: MPJ User Guide</td>
<td>90</td>
</tr>
<tr>
<td>1. Executing and Debugging MPJ Express Programs using the Eclipse IDE</td>
<td>90</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1.1: Amdahl's Law ................................................................. 13
Figure 1.2 Flynn's Taxonomy .......................................................... 16
Figure 1.3 MPJ Design ................................................................. 21
Figure 2.1 Master/Worker Classical Pattern ...................................... 25
Figure 2.2 Master-Worker State Flow ............................................ 26
Figure 2.3 Main Program Flow ..................................................... 27
Figure 2.4 Master-Worker Paradigm Flow: Master as call Master routine and Worker as call Worker routine .................. 28
Figure 2.5 Master Class ............................................................... 33
Figure 2.6 Worker Class ............................................................. 34
Figure 3.1 A Simple Genetic Algorithm ........................................ 44
Figure 3.2 Composition of Genetic Algorithm Classes .................... 46
Figure 3.3 Roulette Wheel .......................................................... 47
Figure 3.4 Crossover Procedure ................................................. 48
Figure 3.5 Single Point Crossover .............................................. 49
Figure 3.6 UML Class Diagram of Simple Genetic Algorithm first version ........ 51
Figure 3.7 UML Class Diagram of Second Simple Genetic Algorithm .......... 55
Figure 3.8 UML Class Diagram of the Third Version of Simple Genetic Algorithm 61
Figure 3.9 UML Class Diagram of Chromosome .............................. 62
Figure 4.1 10-bar plane truss ..................................................... 66
Figure 4.2 25-bar space truss ...................................................... 68
Figure 4.3 Basic Operation Flow ................................................ 71
Figure 4.4 Master Flow ............................................................ 72
Figure 4.5 Worker Flow .................................................................................................................73
Figure 4.6 Simple Genetic Algorithm Flow ....................................................................................74
Figure 4.7 Sequence Diagram PlaneTruss Optimization with SGA .................................................75
Figure 4.8 Sequence Diagram PlaneTruss Optimization with Random Search Method ....................76
Figure 5.1 Result of speed up with Master-Worker in Pipeline method ........................................80
Figure 5.2 Results of speed up with Master-Worker in Update Flow method with 3 new approaches .................................................................................................................................................80
Figure 5.3 Results of the speed up run example 1 on GPU Cluster System .................................82
Figure 5.4 Results of the speed up run example 2 on GPU Cluster System .................................83
Figure 5.5 Results compare speed up between multiple cores with 1, 2, 3 workers and cluster with 23 workers ........................................................................................................................................................................84
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1</td>
<td>Dependency 1</td>
<td>14</td>
</tr>
<tr>
<td>Table 1.2</td>
<td>Dependency 2</td>
<td>14</td>
</tr>
<tr>
<td>Table 1.3</td>
<td>OpenMP Parallel Loop Example 1</td>
<td>18</td>
</tr>
<tr>
<td>Table 1.4</td>
<td>OpenMP Parallel Loop Example 2</td>
<td>18</td>
</tr>
<tr>
<td>Table 1.5</td>
<td>The xdev API</td>
<td>22</td>
</tr>
<tr>
<td>Table 2.1</td>
<td>Master Pseudo Code</td>
<td>29</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Worker Pseudo Code</td>
<td>29</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Pipeline Pseudo Code</td>
<td>30</td>
</tr>
<tr>
<td>Table 2.4</td>
<td>Load Balance Approach</td>
<td>31</td>
</tr>
<tr>
<td>Table 2.5</td>
<td>Block Load Approach</td>
<td>31</td>
</tr>
<tr>
<td>Table 2.6</td>
<td>Max Load Approach</td>
<td>32</td>
</tr>
<tr>
<td>Table 2.7</td>
<td>Master’s Attributes</td>
<td>33</td>
</tr>
<tr>
<td>Table 2.8</td>
<td>Master’s Methods</td>
<td>34</td>
</tr>
<tr>
<td>Table 2.9</td>
<td>Worker’s Attribute</td>
<td>34</td>
</tr>
<tr>
<td>Table 2.10</td>
<td>Worker’s Methods</td>
<td>35</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Random Search Method</td>
<td>37</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>SGA Example 1</td>
<td>42</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>SGA Example 2</td>
<td>43</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>crossover operation</td>
<td>52</td>
</tr>
<tr>
<td>Table 3.5</td>
<td>mutate operation</td>
<td>53</td>
</tr>
<tr>
<td>Table 3.6</td>
<td>evolve operation</td>
<td>54</td>
</tr>
<tr>
<td>Table 3.7</td>
<td>evolve operation</td>
<td>57</td>
</tr>
<tr>
<td>Table 3.8</td>
<td>rollRoulleteWheel</td>
<td>58</td>
</tr>
</tbody>
</table>
Table 3.9 GenomeComparator Class .................................................................59
Table 3.10 doCrossover operation .................................................................63
Table 3.11 doMutate operation ..............................................................64
Table 4.1 Loading conditions for the space truss ........................................68
Table 4.2 Group membership for the space truss ........................................69
Table 4.3 Coordinates of the nodes for the space truss ..............................69
Table 5.1 Comparison of Genetic result with other methods in space truss optimization .................................................................77
Table 5.2 Comparison of Genetic result with other methods in space truss optimization .................................................................78
Table 5.3 Performance Statistic with Intel CPU 4 multiple cores ..............79
Table 5.4 Performance Statistic with GPU Cluster run example 1 ..........81
Table 5.5 Performance Statistic with GPU Cluster run example 2 ..........83
Chapter I
Parallel Computing

1. Introduction

1.1 Background
Parallel computing is when a computation program uses concurrency to decrease the runtime for the solution to a problem as well as increase the scalar of the problem that can be solved. It operates on the principle that large problems can often be divided into smaller ones, which are then solved in simultaneously [1]. There are several different levels of parallelism: job level, program level, inter-instruction level, and intra-instruction level. Parallelism has been used for many years, mainly in high-performance computing [2]. As power consumption by computers has become a concern in recent years, [3] parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors [4].

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism.

1. Multi-core and multi-processor computers have multiple processing elements within a single machine.

2. Clusters, MPPs, and grids use multiple computers to work on the parallelism.

3. Or classification by memory organization such as shared memory, virtual shared memory, distributed memory.

Parallel computer programs are more difficult to write than sequential programs, the reason is that the concurrency in parallel programming introduces several new classes of potential software bugs, whose race conditions are the most common ones. Communication and synchronization between the different subtasks is typically some of the greatest obstacles to have good parallel program performance. Moreover, in parallel computing the hardware is expected to work in excellent condition.

Parallel computing also requires writing a parallel application that follows the flows: from the original problem developers decompose problem into subtasks that’s shared and local
data; then group them into executions unit; finally developers will code with a parallel programming environment [6].

1.2 Amdahl's law and Gustafson’s law

In theoretical way, the speedup from parallelization would be linear that means doubling the number of processors doubles the speed. However, very few parallel algorithms achieve optimal speedup. Most of them have a near linear speedup for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The reduction of execution time, speedup, of a program depends on its properties as a parallel program. It means that the frequency of interaction between the parallel parts of the program can be commensurate with the complexity of the algorithm, and those factors influencing the speedup [6].

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. Amdahl’s law shows that for a given fraction of code capable of parallel or multitasking, that increase of the number of processors does not result in an equivalent increase of speedup. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. If $\alpha$ is the fraction of running time a sequential program spends on non-parallelizable parts, then $S$ is the maximum speed-up with parallelization of the program, which calculated by the following equation:

$$S = \frac{1}{\alpha}$$

If the sequential portion of a program accounts for 10% of the runtime, we can get no more than a 10 times speedup, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."[12]

A different interpretation of Amdahl’s law is Gustafson's law, which is another law in computing, closely related to Amdahl's law [13]. In Gustafson’s law, it states that the speedup with $P$ processors is $S(P)$:

$$S(P) = P - \alpha(P - 1)$$
Figure 1.1: Amdahl's Law

Figure 1.1 shows a graphical representation of Amdahl's law. The speedup of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10 times faster no matter how many processors are used.

1.3 Bernstein conditions
Assumes two subtasks in a parallel program, there are three conditions must be satisfied by two subtasks so that they can be executed in parallel and produce the same results [6]. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. Let S1 and S2 be two subtasks. Bernstein's conditions [14] describe when the two are independent and can be executed in parallel [6]:

\[ \text{Speedup} = \frac{1}{(1 - \frac{P}{100})} \]

where \( P \) is the percentage of the program that can be parallelized.
Consider the following functions, which demonstrate several kinds of dependencies:

```
1: function Dep (a, b)
2:   c: = a·b
3:   d: = 2·c
4: end function
```

Table 1.1 Dependency 1

Operation 3 in Dep (a, b) cannot be executed before (or even in parallel with) operation 2, because operation 3 uses a result from operation 2. It violates condition 1, and thus introduces a flow dependency.

```
1: function NoDep (a, b)
2:   c: = a·b
3:   d: = 2·b
4:   e: = a+b
5: end function
```

Table 1.2 Dependency 2

In this example, there are no dependencies between the instructions, so they can all be run in parallel.

Bernstein’s conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.
2. Parallel Programming

2.1 Introduction

[6] Parallel programming, in contrast to sequential programming, means the performance of several tasks with different data at the same time. Parallel programming is necessary for most scientific simulations if you have high computational requirements. Depending on the complexity of simulation code and the problem being investigated it may take a long time to run on a single processor, or require more memory than is available in a single processor.

Parallel programming is dedicated to performance increase by reducing the run time of an application. One of the most important contributions for achieving this goal is the effective control of the actions which result from parallel working of process in a computer system.

The type of parallel programming used for scientific simulation often depends on the specifics of the parallel operating systems and parallel programming languages available which enable the programmer to initiate the parallel execution of program directly, also to control the communication and synchronization of these in parallel executing programs.

It is possible to exploit small amounts of parallelism using basic thread programming such as that provided by the Java and C programming languages. However, it is often complicated to perform thread programming and it can only exploit a small amount of computing resources. Therefore, if a programmer wants to exploit large amounts of computing resources you need to undertake some level of parallel programming.

There are a range of different types of parallel programming methodologies, often classified by “Flynn’s Taxonomy” as represented in the Figure 1.2.
The dominant programming methodology for most parallel programs currently is MIMD (multiple instructions, multiple data). There are two sub-classes of MIMD, DM (distributed memory), and SM (shared memory). However, it is important to recognize that, in general, MIMD is actually implemented using SPMD (single program, multiple data) methodologies.

For the execution of parallel programs there are currently two main parallel programming approaches in accordance with the type of parallel computer system:

1. Multiprocessing

2. Distributed processing

Multiprocessing means the processors share one common memory during the parallel execution. Additionally, each processor has a local cache. An interconnection network serves as a connection between the processors and the shared memory.

While in distributed processing, each processor has its own local memory. The processor, its local memory, and its communication unit are included in a computer module, for example a personal computer or a work station. Parallel processes which are executed on different computer modules communicate through the exchange of message via a messenger, message transfer system.

Additionally, multiprogramming is a mode of operation on systems with multiple processors as well as with one processor. Several programs are in the memory execute in
parallel that helps the processes to be assigned to the processor quickly and to assure that
the system will be used in the most effective way.

Another important issue of parallel programming is the communication between
processes, because it enables one process to influence another. The communication can be
classified as:

1. Use of shared variables

Moreover, synchronization plays an important role when processes communicate with
each other. For synchronization of two or more processes, one process must perform a
task which is subsequently recognized by another, for example by setting a variable or by
sending a message. Synchronization can be seen as a set of restrictions to the sequence of
occurrence of events that control the communication of processes.

2.2 OpenMP

OpenMP is a library that based on a mode of parallel programming, shared memory
programming. Shared memory method enables processes to have access to some shared
memory space where they can read and write data. Processes communicate with one
another by writing data into the shared memory space for other processes to read.[7]

OpenMP by itself bases on compiler directives which consist of sentinels of the following
form:

Fortran: !$OMP (or C$OMP or *$OMP for F77)

C/C++: #pragma omp

The key words which are instructions to the compiler, for example:

 !$OMP PARALLEL DO

The compiler directives are interpreted by an OpenMP functional compiler and ignored
by a compiler that does not have such functionality. Therefore, it is possible to compile
the same source code to run in both sequence and parallel modes, one of the features
which make it easy to add and test parallel functionality using OpenMP.[7]

There are two main constructs which can be used to provide parallel functionality in
OpenMP:

1. Parallel Loops
2. Parallel Regions

Parallel loops involve distributing the iterations of a standard “for” or “do” loop in C or FORTRAN to different processes, splitting up the work to be done in the loop and thus reducing the overall runtime by using more processes to complete the work of the loop. For instance, the following code does have independent loop iterations [7]:

```c
for (i=0; i<100; i++){
a(i) = a(i) + c(i);
}
```

Table 1.3 OpenMP Parallel Loop Example 1

Whereas this loop does not have independent iterations, as the value of a at one iteration depends on the value of a at the previous iteration:

```c
do i=1,100
   a(i) = a(i-1) + c(i)
end do
```

Table 1.4 OpenMP Parallel Loop Example 2

Beside, parallel regions provide more generic parallel functionality within OpenMP. It points that section of the program contained with the parallel region should be executed in parallel. Moreover, the statements inside the parallel region are executed all the parallel processes, although it is possible to undertake conditional execution depending on the identity of the process[7].
2.3 MPI

MPI is another library that based on distributed memory parallelism. Distributed memory has the ability to use processors that are separate, with no shared memory or shared resources, and only connected by a network. In distributed memory paradigm, processes communication is message passing where the different processes communicate data and synchronize control by sending and receiving message using the network that connects them.

MPI is a set of standards defined by the MPI standards forum which define functions for sending message between processes and associated data types and ancillary functions to support these communications. There are two main standards that make up MPI, MPI-1 and MPI-2. Most basic functionality is provided in the MPI-1 standard, with more advanced features defined in the MPI-2 standard. One of the key objectives of the MPI standard is to provide portability between different parallel machines. Therefore, MPI defines its own data types which are used for data transfers which are then mapped to specific machine data types by the MPI library implementation, which should ensure that programs do not have to be re-written to use different computing hardware.

All communications in MPI take places within a communication spaces called a communicator. The communicator defines the group of processes within the parallel program and provides a mechanism for messages to be sent between them. The MPI library sets up a default communicator when MPI is initialized in a program. This communicator contains all the processes which are running the parallel program, and this is the context within which a process ID can be obtained.

Any MPI program requires at least two library function/sub-routine calls. Init call has a procedure to initialize the message passing functionality and Finalize call finishes it.

C: MPI_Init() MPI_Finalize().
FORTRAN: MPI_INIT() and MPI_FINALIZE().

The two main types of communication mechanism in MPI are:
1. Point-to-point messages
2. Collective message
3. MPJ

3.1 Introduction
In the previous section 2.3, MPI was introduced as the standard for distributed memory in parallel paradigm. The original MPI standard had language bindings for FORTRAN, C and C++. A new generation of distributed, Internet-enabled computing inspired the later introduction of similar message passing APIs for Java [16] [17]. Current implementations of MPI for Java usually follow one of three approaches: use JNI to invoke routines of the underlying native MPI that acts as the communication medium; implement message passing on top of Java RMI (remote method invocation) of distributed objects; or implement high performance MP in terms of low-level “pure” Java communications based on sockets.

The MPJ library described here is developing a next generation MPI for Java, that builds on the lessons learnt from earlier implementations, and incorporating a pluggable architecture that can meet the varying requirements of contemporary scientific computing.

3.2 MPJ Design
MPJ a layered architecture based on an idea of device drivers. The idea is similar to UNIX device drivers, and provides the capability to swap layers in or out as needed. MPJ implements the advanced features of the MPI specifications, which include derived-data types, virtual topologies, different modes of send, and collective communications.

Figure 3 provides a layered view of the messaging system showing the two device levels. The high and base level rely on the MPJ device and xdev level for actual communications. There are two implementations of the mpjdev level. The first uses JNI wrappers to a native MPI library, whereas the other sits on top of xdev. This is a newly proposed Java portability layer that sits between mpjdev (described in earlier works) and the physical hardware. Figure 3 also shows three implementations of xdev, shared memory device (smpdev), Java NIO device (niodev), and GM communications device (gmdev).
Previously, the task of bootstrapping MPI processes over a collection of machines was performed using RSH/SSH based scripts. More recently, runtime infrastructures like MPICH’s SMPD (Super Multi Purpose Daemon) and LAM/MPI’s runtime infrastructure provide an alternative solution.
public static Device newInstance(String dev)
public ProcessID[] init(String[] args)
public ProcessID id()  
public void finish()
public mpjdev.Request isend(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public void send(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public mpjdev.Request issend(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public void ssend(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public mpjdev.Status recv(mpjbuf.Buffer buf, ProcessID srcID, int tag, int context)
public mpjdev.Status probe(ProcessID srcID, int tag, int context)
public mpjdev.Status iprobe(ProcessID srcID, int tag, int context)

Table 1.5 The xdev API

3.3 MPJ implementation

One of the fundamental differences between Java and C is the lack of pointers in Java. Using C, it is possible to use void * to point to any basic data type and/or structures. This is an important characteristic, which forms the basis for efficient messaging. Thus, the first step in developing MPJ was to develop a buffering API that can be used to store all Java basic data types and objects. A reference to this buffer can be passed to the lower level communication devices, which transfers or copies the bytes to the destination.

3.3.1 MPJ buffering API

The derived data types and explicit packing/unpacking for data types is achieved through mpjbuf (MPJ buffering API), which supports the methods like write/read, gather/scatter, and strided-gather/strided-scatter. A buffer object consists of one or more sections that may contain different datatypes. There are two primary sections for a buffer; the static
section of the buffer contains Java primitive data types, whereas; the dynamic section of
the buffer contains Java objects. More details about mpjbuf API can be found in [18].

3.3.2 Point to point communications
MPJ provides blocking and non-blocking point-to-point communications that could be
used to send arrays of basic Java datatypes as well as objects. Also, MPJ provides four
modes of send, which have been defined in the MPI specification document.

3.3.3 Communicators, groups, and contexts
MPI provides higher level abstractions to create parallel libraries, which include
communicators, groups, and contexts. Communicators along with groups provide process
naming; each process in MPI is identified by its rank. The context, which is the attribute
of a communicator, provides a safe communication universe that can be thought of an
additional tag on the messages. Also, there are collective communications on top of point-
to-point communications.

3.3.4 The communication protocols
Implementations of xdev device encapsulate various communication protocols.

Currently, niodev, which is an implementation of xdev using Java NIO, implements two
communication protocols. The eager-send protocol is used by niodev for communicating
small message, typically less than 128 Kbytes. This protocol works on the assumption
that the receiver has got an unlimited device level memory where it can store messages.

There is no exchange of control messages before doing the actual data transmission, thus
minimizing the overhead of control messages that may dominate the total communication
time of small messages. Whenever a send method is called, the sender writes the message
data into the socket channel assuming that the receiver will handle it. At the receiving
side, there can be more than one scenario, depending on whether a matching receive
method is posted by the user or not. If a matching receive method is posted, the message
is copied onto the user specified memory. However, if a matching receive is not posted,
then the message is stored in a temporary buffer. Later, when the receiver is posted, it
copies the message from temporary message to the user specified memory.

The rendezvous protocol is used for large messages, typically greater than 128 Kbytes.
There is an exchange of messages between the sender and the receiver before the actual
transmission of the data payload. The overhead of this exchange of messages is negligible
in terms of the overall communication cost of large messages.
Chapter II
Framework with Master-Worker Paradigm

1. Master-Worker Paradigm Introduction
Master/worker paradigm is a parallel programming method where the master thread distributes the computationally intensive tasks into smaller sub-tasks among worker threads, then waits for the workers threads to complete the allocated tasks, and depending on the problem to be solved, the master then may assign new work out to the workers. Finally, it accumulates the result from sub-results of workers at their completion [19].

The master-worker paradigm is very robust to program. All tasks control is done by one processor, the master. The user should not be burdened with the difficult issue of how to distribute algorithm control information to the various processors. Moreover, the typical parallel programming hurdles of load balancing and termination detection are circumvented. Having a central point of control facilitates the collection of a job’s statistics. Furthermore, a surprising number of sequential approaches to large-scale problems can be mapped naturally to the master-worker paradigm [20].

Programs with centralized control are easily able to adapt to a dynamic and heterogeneous computing environment. If additional processors become available during the course of the computation, they simply become workers and are given portions of the computation to perform. Having centralized control also eases the burden of adapting to a heterogeneous environment, since only the master need be concerned with the matchmaking process of assigning tasks to resources making the best use of the resource characteristics [20].
The two elements of the Master/Worker pattern are the master and the worker. There is only one master, but there can be one or more workers. Logically, the master sets up the calculation and then manages a bag of tasks. Each worker grabs a task from the bag, carries out the work, and then goes back to the bag, repeating until the termination condition is met.

Meanwhile, each worker enters a loop. At the top of the loop, the worker takes a task from the bag of tasks, does the indicated work, tests for completion, and then goes to fetch the next task. This continues until the termination condition is met, at which the master wakes up, collects the results, and finishes the computation.

Master/worker algorithms automatically balance the load. By this, we mean the program does not explicitly decide which task is assigned. This decision is made dynamically by the master as a worker completes one task and accesses the bag of tasks for more work [21].
2. Master-Worker Architecture

2.1 Master Module
The Master component defines the problem domain for a given application. The application domain is broken down into sub tasks that are MPJ enabled. The master need manages a bag of tasks and the number of workers, and then dynamically assigns tasks to the workers and waits. All interaction among the master and worker processes occurs in the form of task and the result entries are exchanged through a communications protocol supported by MPJ. The master/worker paradigm based on MPJ daemon. It also inherently handles all the low-level communication issues.

2.2 Worker Module
The Worker component provides the computational content for the application domain. It first receives the tasks from master and performs the required computation. After sending the results to the Master, the Worker waits for the next task sending from the Master in case any tasks are still available. The Worker repeats the process until there are no more problem tasks left from the Master. It also responds to signals sending from the Master message to start/stop the Worker computation.

The Master and Worker implementations as an abstract class based on MPJ library within application layer. The Workers interacts with the Master also using MPI communication protocols.

![Figure 2.2 Master-Worker State Flow](image-url)
3. Implementation of Framework with Master-Worker Paradigm

The framework implements the master-worker pattern with MPJ as the backbone.

The entire software has been developed in Java to facilitate portability across heterogeneous platforms and to leverage the write once run anywhere features.

Details on the MPJ guide to programming implementation are presented in Appendix A.

3.1 Main Program Flow

![Diagram of Main Program Flow]

Figure 2.3 Main Program Flow
3.2 Master-Worker Flow

Figure 2.4 Master-Worker Paradigm Flow: Master as call Master routine and Worker as call Worker routine
**Master Pseudo Code**

<table>
<thead>
<tr>
<th>Initiate computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive data from call Master routine of main program</td>
</tr>
<tr>
<td>Allocate variables</td>
</tr>
<tr>
<td>Send first tasks to workers</td>
</tr>
<tr>
<td><strong>Repeat</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Until all workers terminated</td>
</tr>
<tr>
<td>Write results to file</td>
</tr>
</tbody>
</table>

**Table 2.1 Master Pseudo Code**

**Worker Pseudo Code**

<table>
<thead>
<tr>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive task from master</td>
</tr>
<tr>
<td>If message contain termination signal</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Else Compute result</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Until done</td>
</tr>
</tbody>
</table>

**Table 2.2 Worker Pseudo Code**
3.3 Improve Master-Work Flow

3.3.1 Improve Master-Worker Parallel Flow with Pipeline Master Methods

Pipeline Master Pseudo Code

Table 2.3 Pipeline Pseudo Code

3.3.2 Improve Master-Worker Flow with 3 Approaches

Load Balance Approach:

In this approach, the Master process divides the number of task equally amongst all the Worker processes. It first sends the values of divided number of task to the worker process. After sending the information associated with the entire tasks is sent to the worker processes, the master component then waits to receive the values of the objective function, new mass value and the maximum violation from all the worker processes.
Loop

Divide number of tasks to equally among number of workers : ncount
Send the number of divided tasks (ncount) to worker
Receive results from worker
Check if tasks is remain
If Yes send next number of tasks to this worker
If No terminate this worker
End loop

Table 2.4 Load Balance Approach

Block Load Approach:

Next, there is a modified approach where sending and receiving takes place one after the other literally on demand. Initially, the master process instructs each worker process to do optimization calculation with each task by sending count of the task. Then it waits to receive the values of the objective function, mass value and the maximum violation from any of the worker processes. If there are more tasks available need to be done, it passes the next task to that worker process. If all the evaluations are completed, it sends a message to that process that no more tasks exit and stop worker process.

Loop

n: number of tasks
Send each tasks from the number of task to workers
n = n - 1
Receive results from worker
Check if tasks is remain
If Yes send next number of task to this worker
If No terminate this worker
End loop

Table 2.5 Block Load Approach
Max Load Approach:

In the previous approaches, the flow is implemented and executed in the master process with only small tasks taken from the number of tasks to run optimization in the worker processes. This approach requires that the number of tasks be sent from the master process to the workers. In this approach, all the processes execute exactly the same number of tasks.

<table>
<thead>
<tr>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>n: number of tasks</td>
</tr>
<tr>
<td>Send n to all the workers</td>
</tr>
<tr>
<td>Wait to receive results from worker</td>
</tr>
<tr>
<td>Terminate this worker</td>
</tr>
<tr>
<td>End loop</td>
</tr>
</tbody>
</table>

Table 2.6 Max Load Approach
3.3 Class Diagram

3.3.1 Master Module

The Master is initiated along with MPI operation provided from MPJ library, receives problem of design optimization that need to be computed from the callMaster of main program.

The Master provides a minimal interface for general services such as setting the number of workers, splitting and packing data, sending data to workers, receiving results and processing results.

Figure 2.5 presents an UML like representation of the Master class. The programmer responsibility is to extend this class and to implement all its abstract methods.

```
<<abstract>>
Master

- np: int
  - nworkers: int
  - id: int
  - start_time: double[]
  - end_time: double[]

+<constructor> Master(nworkers_:int)
+<constructor> Master()
+startMPI(args: String[]): void
+getID(): int
+getNumbersOfWorkers(): int
+startTime(): void
+endTime(): void
+getMPItime(): double[]
+getTime(): double[]
```

Figure 2.5 Master Class

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>np: int</td>
<td>number of processors</td>
</tr>
<tr>
<td>nworkers: int</td>
<td>number of of workers</td>
</tr>
<tr>
<td>id: int</td>
<td>id of master</td>
</tr>
<tr>
<td>start_time: double[]</td>
<td>start time of Master component</td>
</tr>
<tr>
<td>end_time: double[]</td>
<td>end time of Master component</td>
</tr>
</tbody>
</table>

Table 2.7 Master's Attributes
Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master(int nworkers_)</td>
<td>Constructor of Master module</td>
</tr>
<tr>
<td>startMPI(args: String[]): void</td>
<td>start MPI operations and initiate processors</td>
</tr>
<tr>
<td>getID(): int</td>
<td>return the id value of Master module</td>
</tr>
<tr>
<td>getNumbersOfWorkers(): int</td>
<td>return the number of workers for Master module manages</td>
</tr>
<tr>
<td>startTime(): void</td>
<td></td>
</tr>
<tr>
<td>endTime(): void</td>
<td></td>
</tr>
<tr>
<td>getMPITime(): double[]</td>
<td></td>
</tr>
<tr>
<td>getTime(): double[]</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.8 Master's Methods

3.3.2 Worker Module

The Worker class provides common functionality such as connection to master, receiving image together with all necessary parameters, calling algorithm implementation, sending back the result to master and cleaning up worker resources.

Figure 2.63 presents an UML like representation of Worker class. It is programmer responsibility to extend this class and to implement all its abstract methods.

Figure 2.6 Worker Class

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: int</td>
<td>return id of Worker component</td>
</tr>
</tbody>
</table>

Table 2.9 Worker's Attribute
Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker(id_: int)</td>
<td>constructor of Worker component</td>
</tr>
<tr>
<td>getID(): int</td>
<td>return id value of Worker component</td>
</tr>
<tr>
<td>doWork(): void</td>
<td>this abstract methods of main operation of Worker component</td>
</tr>
</tbody>
</table>

Table 2.10 Worker's Methods
Chapter III

Algorithms

1. Random Search

1.1 Introduction
Random search is one of the set of numerical optimization methods that do not require
the gradient of the problem to be optimized. Therefore Random Search can be used on
functions that are not continuous or differentiable. Random Search method was
introduced by Rastrigin [21] who made an early presentation of Random Search along
with basic mathematical analysis. Random Search works by iteratively moving to better
positions in the search space which are sampled from a hyper sphere surrounding the
current position.

1.2 Algorithms Flow
Let \( f: \mathbb{R}^n \rightarrow \mathbb{R} \) is the fitness or cost function which must be minimized.

Let \( x \in \mathbb{R}^n \) designate a position or candidate solution in the search-space. The basic
Random Search algorithm can then be described as:

1. Initialize \( x \) with a random position in the search space.

2. Until a termination criterion is met (e.g. number of iterations performed, or adequate
fitness reached), repeat the following:

   2.1. Sample a new position \( y \) from the hyper sphere of a given radius surrounding
   the current position \( x \) (see e.g. Marsaglia's technique for sampling a hyper sphere.)

   2.2. If \( f(y) < f(x) \) then move to the new position by setting \( x = y \)

3. Now \( x \) holds the best-found position.
1.3 Implementation
Random Search is very easy to implementation, so that we can program Random Search method as a segment of code inline in the operation of Worker component. Table 3.1 gives an example for this.

```java
for (int i = 0; i < components; i++) {
    init = init_recv[i].getDataDouble();
    lower = lower_recv[i].getDataDouble();
    upper = upper_recv[i].getDataDouble();

    temp = Math.random()*(upper - lower);
    if((temp <= upper)&&(temp >= lower)){
        result[i].updateData(temp);
    }
    else{
        result[i].updateData(init);
    }
}
```

Table 3.1 Random Search Method
2. Genetic Algorithm

This section provides a basic coverage on genetic algorithms, describing their functionality in section 2.1. Following that, a simple genetic algorithm (SGA) was developed which will be elaborated in section 2.2. The details of implementation of SGA is described in section 2.3. The SGA was further enhanced to offer more flexibility. This enhanced GA is divided into 2 improvisions for continuous and discrete design value criteria which are described also on section 2.3.

2.1 Introduction of Genetic Algorithm

Genetic algorithms are based on nature and the theory of evolution, therefore belonging to the field of evolutionary computing [22]. Evolutionary computing itself again relies on the Darwinian principles, such as natural selection or survival of the fittest.

The principles of genetic algorithms are quite simple yet powerful. Using a few basic principles – namely reproduction, a fitness function and selection – an encoded problem is solved over the course of several generations of a population composed of individual solutions to the problem. However before we go into detail, let’s cover the historical background from which this approach emerged. Following that, the functionality will be elaborated first by theory, then by example.

2.1.1 Original Ideas

Evolutionary programming was introduced by Lawrence J. Fogel in the USA, while John Henry Holland called his method a genetic algorithm. In Germany Ingo Rechenberg and Hans-Paul Schwefel introduced evolution strategies. From the early nineties on they are unified as different representatives (“dialects”) of one technology, called evolutionary computing. (see [22]).

Initially biologists and geneticists started experimenting with the translation of natural processes into artificial systems in order to further understand the ways of nature, specifically to answer the question how evolution works. The first work was done in 1954 by Nils Aall Barricelli. These early approaches already contained the essentials of the emerging distinct interpretations mentioned before. Later in 1989, Goldberg released a book on genetic algorithms with the title Genetic algorithms in Search, Optimization & Machine Learning [24] which can be viewed as a summary of the research done on genetic algorithms up to then, enhanced with Goldberg’s own research. According to Goldberg “the goals of Holland’s research have been two features:

1. To abstract and rigorously explain the adaptive processes of natural systems.
2. To design artificial systems software that retains the important mechanisms of natural systems.” [24]

2.1.2 Functionality

As mentioned in section 2.1.1, genetic algorithms are based on the important mechanisms of natural systems. But there is a question about what exactly are the important mechanisms.

If one takes a look at the creatures nature created, one will notice nature produced very robust creatures in the course of evolution. Creatures hardly “crash” while processing something like computers do. Goldberg describes robustness as “the balance between efficiency and efficacy necessary for survival in many different environments.” [24] Computer systems, especially operating systems but regular software as well, are also operated in a widely different landscape of hard- and software environments which often cause problems. Such programs could greatly benefit from improved robustness.

The variety of environments attaches importance to one specific feature: adaptability. Higher adaptability means creatures or systems can perform their duty longer and better, bringing more flexibility to the table [24]. Eventually, software could grow into something new by adapting to changes. The change is one ever-present constant; else there would be no need for new software in the same old branches over and over again. One ultimate system would be enough to satisfy the needs of the current and all coming generations. Nevertheless since this is not the case, adaptability is the key to the longevity and usefulness of software.

To understand how a genetic algorithm works we will take a look at its prototype: nature.

The following example was taken from Jean-Philippe Rennard [25] and slightly modified for the purpose of clarity:

“Imagine the prehistoric ancestor of whales, the basilosaurus. It was about 15 meters long and weighted 5 tons. It had a quasi-independent head and posterior paws. It moved using undulatory movements and hunted small prey.

Movement in a viscous element like water is very hard and requires big effort. To hunt, the basilosaurus must have had enough energy to move, control its trajectory and snatch its prey. The anterior members of basilosaurus were not really adapted to swimming. To adapt them, a double phenomenon must occur: the shortening of the “arm” with the locking of the elbow articulation and the extension of the fingers which will constitute the base structure of the flipper.”
Through time, subjects appeared with longer fingers and shorter arms. They could move faster and more precisely than before, and therefore, live longer and have many descendants. Meanwhile, other improvements occurred concerning the general aerodynamic like the integration of the head to the body, improvement of the profile, strengthening of the caudal fin, etc. Finally producing a subject perfectly adapted to the constraints of an aqueous environment. (see [25])

This is the basic principle behind genetic algorithms: The higher the fitness of a basilosaurus, the longer it will live. Whereas higher adaptability helps to react to new circumstances more rapidly, increasing its fitness. A basilosaurus with higher fitness is more likely to attract a partner, which leads to descendants that will carry on his genes. Eventually the improved basilosaurus genes with short arms and long fingers will prevail. But adaption can only take place between two generations. Existing basilosaurus won’t be able to grow short arms and long fingers within their lifetime, but their offspring may be born possessing those features.
2.1.2.1 Three Basic Operators

All creatures have genes stored in the form of DNA that consists of strings which encode short groups of linked features. Let’s consider such a DNA string as a binary string, whereas the position of a binary digit indicates its meaning (e.g. paw or fingers) and the value expresses which kind of state it possesses (e.g. 0 = short, 1 = long). By combining those properties, we are able to express features like short paw or long finger.

In crossover process a creature couple procreates their offspring. The offspring is composed by a mix of the features of their mother and their father or by completely copying the identical features of one parent. Crossover allows for certain creatures features – or groups of features – to be combined in a possibly new way which might be more useful in the present or future environment.

However, crossover does not allow the population of creatures as a whole to develop new features since it’s only a recombination of existing features. The development of new features is taken care of by mutation. As mentioned above, we can extract the three following basic operators:

1. Reproduction
2. Crossover
3. Mutation

These three operators constitute the essential functionality of any genetic algorithm.
2.1.2.2 Understand Genetic Algorithm by Example

For the example, find a maximum for the function $f(x) = x^2$ with $0 \leq x \leq 15$.

In the beginning, it needs to encode $x$ in an appropriate way. A binary string representing a binary number will be enough in this case. The binary number starts with the most significant bit and has a maximum of four digits. Consider the following population, containing four individual chromosomes and their corresponding fitness:

<table>
<thead>
<tr>
<th>Code Value</th>
<th>Fitness</th>
<th>Decoded value</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1100</td>
<td>0.8</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>B 1010</td>
<td>0.6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>C 0011</td>
<td>0.2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D 0000</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Generation: 1  Sum: 1.6

**Table 3.2 SGA Example 1**

The fitness function for this example is pretty simple. It’s calculated by decoding the binary number into a decimal number $k$, which is used in the following formula:

$$fitness = \frac{k}{\text{max}(x)} = \frac{k}{15}$$

Next, to generate the next population the individuals of our current generation have to reproduce: Two individuals will be randomly chosen as mating partners – though the random selection is influenced by the fitness value: Individuals with higher fitness should possess a higher probability of being selected.

We select individual A and B to mate. To create the offspring, we will need to determine a random crossover site, let’s say we rolled $x$ position = 2:

A 1 1 x 0 0

B 1 0 x 1 0

Beginning with child A’, all the bits of parent A are transferred to the first child until the crossover site is met. Then the remaining bits starting from the crossover its of parent B are transferred to the child A’. Likewise, but vice versa, we transfer all bits from parent B
until the x position to child B’, then transfer all bits beginning at the x position from parent A to child B’.

Incidentally, after crossing a mutation happens in child B’, randomly flipping the bit at position 4. The offspring A’ and B’ looks like this:

A’ 1 1 1 0
B’ 1 0 0 1 without mutation: 1 0 0 0

The next children are produced using B and C as parents. After this, we have a population of four individuals, meaning no further reproduction is necessary.

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Fitness</th>
<th>Decoded value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’</td>
<td>1110</td>
<td>0.9</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>B’</td>
<td>1001</td>
<td>0.6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>C’</td>
<td>1011</td>
<td>0.7</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>D’</td>
<td>0010</td>
<td>0.1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Generation: 2 Sum: 2.3

Table 3.3 SGA Example 2

From above, the individual D completely dropped out since it was not selected or mating because of its low fitness value. Also, the best fitness increased from 0.8 to 0.9, as well as the sum of fitness which increased from 1.2 to 2.3.

2.2 A Simple Genetic Algorithm

Genetic algorithms (GAs) are adaptive, global search procedures that are designed to minimize the underlying processes governing natural selection and evolution. A GA works on a population of individuals to identify relatively better individuals among the population, and combines the information embedded in the individuals to form newer individuals that are generally better. Each individual represents a solution to an optimization problem and the fitness of the solution is evaluated based on the performance of that solution in solving the given optimization problem. Through repeated application of a set of basic GA operators, including selection, mutation, and crossover, on the population of the individuals, the population generally converges to a solution that performs well with respect to the evaluation criterion. Although GAs do not guarantee
global optima, in practice a population based search is robust as compared to local search approaches, and has been demonstrated to be quite successful in finding good solutions in an efficient manner. The following figure briefly illustrates a simple steady state GA procedure.

Figure 3.1 A Simple Genetic Algorithm
The objectives of this research are to:

1. Develop GA object classes and classes that represent a generic optimization model for generic use

2. Implement an interface that helps user to approach structural design optimization problems and solves them by selecting from a combination of various decision variable representations, GA operators, and various approaches for handling constraint violations;

3. Demonstrate the use of the above interface for solving standard structural design optimization problems

4. Assist users in understanding and exploring the working of GAs and in learning about GAs.

2.2.1 Design and Operations

2.2.1.1 Data Structure
Designing appropriate data structures to represent the hierarchy of genetic information content is of central importance to the GA search procedure. It is also essential to keep the design modular and sufficiently abstract so that it facilitates flexibility for further development.

Typically, the hierarchy can be represented adequately by a gene at the lowest level. A chromosome (or genome) is a vector of similar types of genes. An individual is a container for a set of chromosomes. The information necessary for evaluation of the fitness properties is encapsulated within the individual. A population comprises of a set of individuals and is the basic evolving unit. Individuals in a population undergo crossover and mutation to produce the next generation of individuals. Figure 2.2 depicts the composition of the various GA classes. The ovals represent the classes and the nested structure reflects the composition relationship amongst the classes.
The links represent the association relationships between the classes. A unidirectional link, i.e., an arrow, indicates that the association relationship is one sided. In this case, Chromosome is considered “visible” to Evaluation Function.

2.2.1.2 Design Variable Representations

The representation of a potential solution by an individual is the manner in which the genetic material is coded. Defining an appropriate representation is part of the art of using GAs. The representation should be minimal but completely expressive. The user should select a representation so that short, low-order schemata are relevant to the underlying problem and relatively unrelated to schemata over other fixed positions [10].

The current Simple Genetic Algorithm implementation supports binary (b) representations for decision variables. In the next improvement version of Simple Genetic Algorithm, we support set of double variables (d) which can be represented either as a double array for real design value. Integer programming problems can be formulated by specifying the real decision variable to be of type d or of type b where the number in the parentheses indicates the number of decimal digits desired.

These representations are adequate for solving function optimization problems such as for discrete design variable or continues design variable. Different problems, however, need different representations and these can be easily coded by the user by writing a class that implements the interface of appropriate Simple Genetic Algorithm classes. The primary
implemented operations are `crossover()`, `mutate()` and `getFitnessFunction()`. In which the operation `getFitnessFunction` is abstract and will be implemented depend on the structural optimization problem.

### 2.2.1.3 Selection Schemes

The selection operator ensures that the GA yields incrementally better `individuals` in each generation. The `individuals` with larger fitness values get higher probability of being selected for mating than `individuals` having low fitness values. The selected `individuals` are then do `crossover` to reproduce new `individuals` of next generation. The following selection schemes are supported:

**Roulette Wheel**

In this type of selection, the probability of an individual being selected is proportional to its relative fitness.

\[
P_{\text{select}} = \frac{f_i}{\sum_{i=1}^{N} f_i}
\]

where

\[f_i = \text{fitness of individual } i\]

\[N = \text{population size}\]

This ensures that individuals with higher fitness have a better representation to be chosen to do crossover.

### 2.2.1.4 Crossover Operators

The `crossover` operator defines the procedure for generating child `individual(s)` by mating two parent `individuals`. A `crossover` probability determines if the parent `individuals` will cross to produce child `individuals` and the level of mixing of genetic information.

Convergence is rapid with higher `crossover` probabilities and may lead to premature convergence. The ideal `crossover` probability is problem dependent. `Individuals` may consist of multiple chromosomes, in which case the corresponding `chromosomes` of the
two parent individuals P1 and P2 undergo crossover. Figure 3.4 illustrates the \textit{crossover} procedure.

\textbf{Procedure ‘Crossover’}
1. Select two parent individuals, \( P_1 \) and \( P_2 \), for crossover.
2. if\( (\text{Crossover}) \) then
   \[
   \text{Do } (P_1[\text{Chromosome}_i] \times P_2[\text{Chromosome}_i]) \quad \forall i = 1, N_c
   \]
   \text{where } N_c \text{ is the number of chromosomes in an individual}
3. else
   \text{Place } P_1 \text{ and } P_2 \text{ in the next generation.}

\textbf{Figure 3.4 Crossover Procedure}

In the current implementation, the following \textit{crossover} operators are supported.

\textbf{Single Point Crossover}

A single crossover point on both parents’ organism strings is selected. All data beyond that point in either organism string is swapped between the two parent organisms. The resulting organisms are the children:
2.2.1.5 Mutation Operators

The mutation operator defines the procedure for mutating each chromosome. Mutation introduces new genetic material into the gene pool of the population of individuals. This may result in the exploration of previously unexplored points in the decision space or reintroduce lost genetic material. The mutation operator contributes to the global search by enabling the procedure to explore potentially new parts of the decision space as well as to maintain population diversity. The mutation is controlled by a mutation probability that determines if a particular chromosome undergoes mutation.

Larger mutation probability increases the probability of disruption of good schemata, but increases population diversity. The ideal mutation rate depends on the characteristics of the decision and objective spaces.

Mutation is different for various data types. For example, a typical mutation for a binary gene is flipping the bit with a given probability. In the current Simple Genetic Algorithm, the following mutation operators are supported.

Random change mutation

This operator changes the value of a randomly chosen set of genes. The number of genes in this set can be specified by the user. For example, for a real number gene, this operator will randomly set the value of this real number to be a number between the minimum and maximum bounds of that real number gene. This mutation operator is suitable for chromosomes of real number genes and genes having more complex data structures.
2.3 Implementation of Simple Genetic Algorithm

2.3.1 First Version of Simple Genetic Algorithm

The first version of Simple Genetic Algorithm which described in this section treats genes as a single bit; while you can consider a gene to be an arbitrary data structure, the approach of using single bit genes and specifying the number of genes (or bits) in a chromosome is very flexible. A population is a set of chromosomes. A generation is defined as one reproductive cycle of replacing some elements of the chromosome population with new chromosomes produced by using a genetic crossover operation followed by optionally mutating a few chromosomes in the population.
The constructors build an array of integer rouletteWheel which is used to weight the most fitness chromosomes in the population for being the parents of crossover operations. When a chromosome is being chosen, a random integer is selected that is used as an index into the rouletteWheel array; the values in the array are all in the range of \([0, \text{number of genes per chromosome} – 1]\). More fit chromosomes are heavily weighted in favor of being chosen as parents for the crossover operations. The algorithm for the crossover operation is fairly simple; here is the implementation:
public void crossover() {
    int num = (int) (numChromosome * probCrossover);
    for (int i = num - 1; i >= 0; i--) {
        int c1 = 1 + (int) ((roulette.getRouletteWheelSize() - 1) * Math.random() * 0.9999f);
        int c2 = 1 + (int) ((roulette.getRouletteWheelSize() - 1) * Math.random() * 0.9999f);
        c1 = roulette.getRouletteWheelSlot(c1);
        c2 = roulette.getRouletteWheelSlot(c2);
        if (c1 != c2) {
            int locus = 1 + (int) ((numGeneOfChrom - 2) * Math.random());
            for (int g = 0; g < numGeneOfChrom; g++) {
                if (g < locus) {
                    setGene(i, g, getGene(c1, g));
                } else {
                    setGene(i, g, getGene(c2, g));
                }
            }
        }
    }
}

Table 3.4 crossover operation

The class variable probCrossover is used to calculate the number of chromosomes in the population will be replaced by the results of crossover reproduction. The array of chromosomes has been sorted in decreasing order of fitness before this method is called, so the least fit individuals are at the higher array indices; these individuals will be replaced. Two chromosome indices c1 and c2 are calculated using a random number generator and the rouletteWheel array.
The index locus is a random value in the range \([1, \text{number of bits per chromosome} - 2]\). It is important to note that all indexing is zero based. Genes at index less than locus are copied from the beginning of the chromosome indexed by \(c1\) and genes at index greater than or equal to locus are replaced by the genes at the end of the chromosome indexed by \(c2\).

The algorithm for mutating a chromosome is simple: randomly choose a gene and flip its value.

```java
public void mutate() {
    int num = (int) (numChromosome * probMutation);
    for (int i = 0; i < num; i++) {
        int c = 1 + (int) ((numChromosome - 1) * Math.random() * 0.99);
        int g = (int) (numGeneOfChrom * Math.random() * 0.99);
        setGene(c, g, !getGene(c, g));
    }
}
```

The method sort re-orders both the chromosome array and the associated fitness array in decreasing order of fitness. The top level control method in class Genetic is evolve. This method is called once per generation and performs the following actions:

1. Calculates the fitness of the chromosomes in the population by calling the abstract method
2. Call calcFitness
3. Sort the chromosomes in decreasing order of fitness by calling method sort
4. Perform genetic crossover reproductions by calling method doCrossOvers
5. Perform genetic mutations by calling method doMutations
public void evolve() {
    crossover();
    mutate();
    removeDuplicates();
    calcFitness();
    sort();
}

Table 3.6 evolve operation
2.3.2 Second Version of Simple Genetic Algorithm

The second version of Simple Genetic Algorithm which described in this section treats genes as an array of double values and calls it Genome instead of chromosome as the first version; while you can consider a gene to be an arbitrary data structure. A population is a set of genomes. In this Simple Genetic Algorithm, there are 2 populations: thisGeneration and nextGeneration; where nextGeneration is defined as one reproductive cycle of replacing some elements of the genome population with new genomes produced by using a genetic crossover operation followed by optionally mutating a few chromosomes in the thispopulation.

Class Diagram

![UML Class Diagram of Second Simple Genetic Algorithm]

Figure 3.7 UML Class Diagram of Second Simple Genetic Algorithm
The algorithm code contains two simple classes, SimpleGeneticAlgorithm and Genome, plus a helper class GenomeComparator.

The Genome class can be thought of as a simple container. The user is expected to take these values and scale them to whatever values they require. Since mutation occurs on the genome, the mutate method is found in this class. The crossover operator requires access to the private data of the Genome, so it is also a member function which takes a second Genome, and outputs two child Genome objects. The fitness of a particular genome is also stored within the Genome object. There are some additional helper functions that maybe found in the code itself.

The SimpleGeneticAlgorithm class does all the work. The genetic algorithm consists of the following basic steps:

1. Create a new population

2. Select two individuals from the population weighting towards the individual that represents the best solution so far.

3. 'Breed' them to produce children.

4. If we don't have enough children for a new population return to step 2.

5. Replace old population with new.

6. If we have not produced enough generations return to step 2.

7. We have finished.
public void evolve(){
    if(genome_size_ == 0){
        throw new ArrayIndexOutOfBoundsException("Genome size is not set");
    }
    /* Create the fitness table */
    fitnessTable = new ArrayList<Double>();
    thisGeneration = new ArrayList<Genome>(generation_size_);
    nextGeneration = new ArrayList<Genome>(generation_size_);
    creatGenome(mutation_rate_, high_, low_);
    sortPopulation();
    boolean is_write = false;
    if(str_fitness_ != ""){
        is_write = true;
        System.out.println(str_fitness_);
    }
    for(int i = 0; i < generation_size_; i++){
        createNextGeneration();
        sortPopulation();
        if(is_write){
            double d;
            d = (double)((Genome)thisGeneration.get(population_size_ - 1)).getFitness();
            System.out.println("[" + i + ", [" + d + "]");
        }
    }
}

Table 3.7 evolve operation
When selecting individuals to breed, we use what is called the *rollRouletteWheel* method. This is where fitter individuals have a larger proportion of the 'wheel' and are more likely to be chosen. We chose to store the finesses cumulatively in *ArrayList* as it had some nice features like sorting. Unfortunately, its binary search method was only for exact values, so we had to implement the following work around:

```java
private int rollRouletteWheel()
{
    double randomFitness = random_generator.nextDouble()*total_fitness;
    int idx = -1;
    int mid;
    int first = 0;
    int last = population_size - 1;
    mid = (last - first)/2;
    while(idx == -1 && first <= last)
    {
        if(randomFitness < (double)fitnessTable.get(mid)){
            last = mid;
        } else if(randomFitness > (double)fitnessTable.get(mid)){
            first = mid;
        }
        mid = (first + last)/2;
        /* Value lies between i and i+1 */
        if((last - first) == 1){
            idx = last;
        }
    }
    return idx;
}
```

Table 3.8 rollRouletteWheel
The GenomeComparator class inherits from the Comparator interface. Each generation is stored in ArrayList, and we wish to sort each generation in order of fitness. We therefore need to implement this helper class as described in Table 3.9:

```java
public class GenomeComparator implements Comparator<Genome> {
    @Override
    public int compare(Genome o1, Genome o2) {
        if(o1.getFitness() > o2.getFitness()){
            return 1;
        }
        else if(o1.getFitness() == o2.getFitness()){
            return 0;
        }
        else{
            return -1;
        }
    }
}
```

Table 3.9 GenomeComparator Class
2.3.3 Third Version of Simple Genetic Algorithm

The third version of Simple Genetic Algorithm (SGA) which described in this section treats genes as an array of double values and calls it Genome instead of chromosome as the first version. And improved from the second version, in the third version of Simple Genetic Algorithm, there is a new concept was added, that is Chromosome, Chromosome here have an array of Genome, that help with the case of multiple criteria of design values. In case of multiple design value, each Genome will represent a criterion and Chromosome will be the set of all the design values criteria. A population is a set of chromosomes. Similar to the second version SGA, there are 2 populations: thisGeneration and nextGeneration; where nextGeneration is defined as one reproductive cycle of replacing some elements of the chromosomes population with new chromosomes produced by using a genetic crossover operation followed by optionally mutating a few chromosomes in the thispopulation.

Class Diagram
All the components and operations of the third version of SGA are exactly the same with the second version of SGA, except there are some little changes in classes: SimpleGeneticAlgorithm, ChromosomeComparator and a new class was implemented: Chromosomes:
Figure 3.9 UML Class Diagram of Chromosome
The *Chromosome* class can be thought of as a simple container of *Genome* array. The user is expected to take these values and scale them to whatever values they require. Since mutation occurs on the *Genome*, the *mutate* method is also implemented in this class. The *crossover* operator requires access to the private data of the *Genome*, so it is also a member function which takes a second *Genome*, and outputs two child *Genome* objects, then the crossover operation implemented in *Chromosome* class to keep that, each *Genome* will do the crossover with appropriate *Genome* in the second *Chromosome*, then all *Genome* childs will be copy to *Chromosome* childs.

```java
public Chromosome[] doCrossover(Chromosome chromosome2) throws Exception{
    /* Init 2 chromosome childs */
    Chromosome[] childs = new Chromosome[2];
    childs[0] = new Chromosome(genometype_number_, genome_size_, high_,
                             low_, mutation_rate_);
    childs[1] = new Chromosome(genometype_number_, genome_size_, high_,
                             low_, mutation_rate_);
    /* In cross-over, each genome do cross over with match up genome in
    * 2 chromosome parents then copy to childs*/
    for(int i = 0; i < genometype_number_; i++){
        Genome[] temp = new Genome[2];
        Genome genomel = chromosomes[i];
        Genome genome2 = chromosome2.getGenome(i);
        temp = genomel.doCrossover(genome2);
        childs[0].setGenome(temp[0], i);
        childs[1].setGenome(temp[1], i);
    }
    return childs;
}
```

Table 3.10 doCrossover operation
public void doMutate(double[] high, double[] low){
    for(int i = 0; i < genometype_number_; i++){
        chromosomes[i].doMutate(high[i], low[i]);
    }
}

Table 3.11 doMuate oeration
Chapter IV

Apply Framework to Structural Design Optimal Problem Domains

1. Introduction
This chapter focuses on the use of a search technique called Genetic Algorithm (GA) to optimize the design of plane and space trusses. This technique considers both discrete and continuous design value search space, so it is not only yields more realistic results than linear programming methods, but also has problem independent characteristic. This means that the code developed for designing trusses can be reused to solve the remaining framed structures (plane and space frames, plane grids and frames) with little change.

Beside this chapter also shows how the Random Search Method is used in structural design optimization, but it’s really easy to see that Random Search Method is just implement as a segment of code, which is inlined in the implementation of Master-Worker paradigm. Refer the section 3.1 for more details.

2. Structural Model Examples

2.1 Optimization of a Plane Truss
The first example is the 10-bar plane truss taken from Rajeev and Krishnamoorthy [26], shown in Figure 4.1. Define of a plane truss [27]

"A plane truss is idealized as a system of members lying in a plane and interconnected at hinged joints. All applied forces are assumed to act in the plane of the structure, and all external couples have their moment vectors normal to the plane. The loads may consist of concentrated forces applied to the joints, as well as loads that act on the members themselves. For purposes of analysis, the latter loads may be replaced by statically equivalent loads acting at the joints. Then the analysis of a truss subjected only to joint loads will result in axial forces of tension and compression in the members. In addition to these axial forces, there will be bending moments and shear forces in those members having loads that act directly upon them. The determination of all such stress resultants constitutes the complete analysis of the forces in the members of a truss."
Now, let’s consider a 10-bar truss optimization problem taken from Rajeev and Krishnamoorthy [28], shown in Fig. 4.1. The objective function of the problem is to minimize the weight of the structure, $f(x)$,

$$f(x) = \sum_{i=1}^{10} \rho A_i L_i$$

where $x$ is the candidate solution, $A_i$ is the cross-sectional area of the $i$th member, $L_i$ is the length of the $i$th member, and $\rho$ is the weight density of the material. The assumed data are: modulus of elasticity, $E = 10^4$ ksi ($6.89 \times 10^4$ MPa), $\rho = 0.10$ lb/in$^3$ (2,770 kg/m$^3$), and vertically downward loads of 100 kips (445.374 kN) at nodes 2 and 4. Additionally, the truss is subject to the following set of constraints

$$\sigma_i \leq \sigma_x, \text{ for } i = 1 \text{ to } 10$$
$$u_i \leq u_x$$
where $\sigma_i$ is the stress in member $i$, $\sigma_a$ is the maximum allowable stress for all members, $u_i$ is the displacement of each node (horizontal and vertical), and $u_a$ is the maximum allowable displacement for all nodes. These constraints can be expressed in normalized form as

\[
\frac{\sigma_i}{\sigma_a} - 1 \leq 0; \quad \frac{u_i}{u_a} - 1 \leq 0
\]

The fitness function used was

\[
F(x) = \frac{1}{f(x)[1000\nu + 1]}
\]

where $\nu$ is a counter that keeps track of the number of constraints violated by a given solution. We can easily see how when there is no violation to the constraints, the fitness function returns simply the inverse of the weight (this is necessary because the GA only maximizes). As constraints are violated, the fitness is lowered correspondingly. The constant 1000 was determined experimentally.

The additional constraints for this problem are the following: the maximum displacement is 2 inches (50.8 mm) and the stresses are limited to ±25 ksi (172.25 MPa). The list of possible cross sections, taken from the American Institute of Steel Construction Manual [11], is $S=\{1.62, 1.80, 1.99, 2.13, 2.38, 2.62, 2.63, 2.88, 2.93, 3.09, 3.13, 3.38, 3.47, 3.55, 3.63, 3.84, 3.87, 3.88, 4.18, 4.22, 4.49, 4.59, 4.80, 4.97, 5.12, 5.74, 7.22, 7.97, 11.5, 13.5, 13.9, 14.2, 15.5, 16.0, 16.9, 18.8, 19.9, 22.0, 22.9, 26.5, 30.0, 33.5\} (\text{in}^2)$.

2.2 Optimization of a Space Truss

The second example is the 25-bar space truss taken from Rajeev and Krishnamoorthy [20-26], shown in Figure 3.2. A space truss is defined in Gere and Weaver [8-28] as follows:
"A space truss is similar to a plane truss except that the members may have any directions in space. The forces acting on a space truss may be in arbitrary directions, but any couple acting on a member must have its moment vector perpendicular to the axis of the member. The reason for this requirement is that a truss member is incapable of supporting a twisting moment."

Loading conditions for this space truss, member groupings, node coordinates are given in Table 4.1, Table 4.2, and Table 4.3 respectively. The assumed data are: modulus of elasticity, $E = 10^4$ ksi ($6.89 \times 10^4$ MPa), $\rho = 0.10$ lb/in$^3$ (2770 kg/m$^3$); $\sigma_a = \pm 40000$ psi, $u_a = \pm 0.35$ in. The set of available areas is $S = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.8, 3.0, 3.2, 3.4\}$ (in$^2$).

<table>
<thead>
<tr>
<th>Node</th>
<th>Fx(lbs)</th>
<th>Fy(lbs)</th>
<th>Fz(lbs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>-10000</td>
<td>-10000</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-10000</td>
<td>-10000</td>
</tr>
<tr>
<td>3</td>
<td>500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>600</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1 Loading conditions for the space truss
<table>
<thead>
<tr>
<th>Group Number</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-2</td>
</tr>
<tr>
<td>2</td>
<td>1-4, 2-3, 1-5, 2-6</td>
</tr>
<tr>
<td>3</td>
<td>2-5, 2-4, 1-3, 1-6</td>
</tr>
<tr>
<td>4</td>
<td>3-6, 4-5</td>
</tr>
<tr>
<td>5</td>
<td>3-4, 5-6</td>
</tr>
<tr>
<td>6</td>
<td>3-10, 6-7, 4-9, 5-8</td>
</tr>
<tr>
<td>7</td>
<td>3-8, 4-7, 6-9, 5-10</td>
</tr>
<tr>
<td>8</td>
<td>3-7, 4-8, 5-9, 6-10</td>
</tr>
</tbody>
</table>

Table 4.2 Group membership for the space truss

<table>
<thead>
<tr>
<th>Node</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-37.50</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>37.50</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>-37.50</td>
<td>37.50</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>37.50</td>
<td>37.50</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>37.50</td>
<td>-37.50</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>-37.50</td>
<td>-37.50</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>-100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>-100</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>-100</td>
<td>-100</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.3 Coordinates of the nodes for the space truss
3. Implementation

For structural optimization method, both Random Search Method and Simple Genetic Algorithm are used, but mainly focus to Simple Genetic Algorithm. The stopping criteria were always a maximum number of generations, and the crossover and mutation probabilities are chosen as 0.80 and 0.05, respectively.

The analysis of each truss was performed using the programs from miniFE with minimal package *inf-minife-minimal.jar*, in which includes only the basic operation of the FEM structural analysis.

Basic operation is simple: The set of all possible cross section areas is fed to program from file that stores cross section values, in case of continuous cross section values; user will directly input the range of design value into program. Then constrains on max stress and max displacement are provided. The program does structural analysis on plane truss in models class that implemented in Java class of model package (PlanTruss.java, SpaceTruss.java). Next, Simple Genetic Algorithm is called and parallel executed in master-worker paradigm with number of generation, population size, length of chromosome, etc. In each generation generated by the Worker, they will be fetched to truss models to analysis then result including new mass value, fitness value of generation and number of violation constrains which will be send to the Master. Master print out the result, and then do the analysis to find best generation that fits to model based on fitness value and mass value.
Basic Operation Flow

Figure 4.3 Basic Operation Flow
Master Flow

Figure 4.4 Master Flow
Worker Flow

Figure 4.5 Worker Flow
Simple Genetic Algorithm Flow

Figure 4.6 Simple Genetic Algorithm Flow
4. Sequence Diagram of Structural Optimization

4.1 PlaneTruss Optimization with Simple Genetic Algorithm

![Sequence Diagram PlaneTruss Optimization with SGA]

Figure 4.7 Sequence Diagram PlaneTruss Optimization with SGA
4.2 PlaneTruss Optimization with Random Search Method

Figure 4.8 Sequence Diagram PlaneTruss Optimization with Random Search Method
Chapter V

Results and Comparisons

1. Space Truss

<table>
<thead>
<tr>
<th>Method</th>
<th>Weight</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rizz</td>
<td>545.16</td>
<td>0.01</td>
<td>1.988</td>
<td>2.991</td>
<td>0.01</td>
<td>0.01</td>
<td>0.684</td>
<td>1.676</td>
<td>2.662</td>
</tr>
<tr>
<td>Schmit</td>
<td>545.22</td>
<td>0.01</td>
<td>1.964</td>
<td>3.033</td>
<td>0.01</td>
<td>0.01</td>
<td>0.64</td>
<td>1.68</td>
<td>2.67</td>
</tr>
<tr>
<td>Rajeev</td>
<td>546.01</td>
<td>0.1</td>
<td>1.8</td>
<td>2.3</td>
<td>0.2</td>
<td>0.1</td>
<td>0.8</td>
<td>1.8</td>
<td>3</td>
</tr>
<tr>
<td>Zhu</td>
<td>562.93</td>
<td>0.1</td>
<td>1.9</td>
<td>2.6</td>
<td>0.1</td>
<td>0.1</td>
<td>0.8</td>
<td>2.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Genetic</td>
<td>272.9</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.3</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 5.1 Comparison of Genetic result with other methods in space truss optimization

All the other methods mentioned here, related to the space truss optimization, reported in the literature for the space truss. For more details information about each one of these methods, refer to [27] and [35]
2. Plane Truss

<table>
<thead>
<tr>
<th>Method</th>
<th>Weight</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTDYN</td>
<td>5472</td>
<td>25.7</td>
<td>0.1</td>
<td>25.11</td>
<td>19.39</td>
<td>0.1</td>
<td>0.1</td>
<td>15.4</td>
<td>20.32</td>
<td>20.74</td>
<td>1.14</td>
</tr>
<tr>
<td>CONMIN</td>
<td>5563</td>
<td>25.2</td>
<td>1.89</td>
<td>24.87</td>
<td>15.83</td>
<td>0.1</td>
<td>1.75</td>
<td>16.76</td>
<td>19.73</td>
<td>20.98</td>
<td>2.51</td>
</tr>
<tr>
<td>Rajeev</td>
<td>5613.84</td>
<td>33.5</td>
<td>1.62</td>
<td>22</td>
<td>15.5</td>
<td>1.62</td>
<td>1.62</td>
<td>14.2</td>
<td>19.9</td>
<td>19.9</td>
<td>2.62</td>
</tr>
<tr>
<td>M-3</td>
<td>5719</td>
<td>25.84</td>
<td>3.07</td>
<td>26.42</td>
<td>12.77</td>
<td>0.1</td>
<td>3.43</td>
<td>19.34</td>
<td>19.17</td>
<td>18.76</td>
<td>4.42</td>
</tr>
<tr>
<td>M-5</td>
<td>5725</td>
<td>25.83</td>
<td>2.88</td>
<td>26.45</td>
<td>12.75</td>
<td>0.1</td>
<td>3.77</td>
<td>19.37</td>
<td>19.18</td>
<td>18.77</td>
<td>4.38</td>
</tr>
<tr>
<td>GRP-UI</td>
<td>5727</td>
<td>24.78</td>
<td>4.17</td>
<td>24.78</td>
<td>14.45</td>
<td>0.1</td>
<td>4.17</td>
<td>17.46</td>
<td>19.26</td>
<td>19.27</td>
<td>5.26</td>
</tr>
<tr>
<td>SUMT</td>
<td>5932</td>
<td>30.69</td>
<td>2.37</td>
<td>31.62</td>
<td>11.66</td>
<td>0.1</td>
<td>3.71</td>
<td>21.71</td>
<td>20.9</td>
<td>13.97</td>
<td>3.26</td>
</tr>
<tr>
<td>LINRM</td>
<td>6249</td>
<td>21.57</td>
<td>10.98</td>
<td>22.08</td>
<td>14.95</td>
<td>0.1</td>
<td>10.98</td>
<td>18.91</td>
<td>18.42</td>
<td>18.4</td>
<td>13.51</td>
</tr>
<tr>
<td>Genetic</td>
<td>3833.2</td>
<td>3.87</td>
<td>1.8</td>
<td>7.22</td>
<td>2.13</td>
<td>1.8</td>
<td>1.8</td>
<td>3.47</td>
<td>2.13</td>
<td>2.88</td>
<td>3.84</td>
</tr>
</tbody>
</table>

Table 5.2 Comparison of Genetic result with other methods in space truss optimization

All the other methods mentioned here, related to the plane truss optimization, reported in the literature for the plane truss. For more details information about each one of these methods, refer to [27] and [34]
3. Performance

The results were showed from Table 5.3. The parallel speedup on each computing platform has been measured using the MPI time. The measure obtained on two computing platform: the computer has an Intel CPU with 4 multiple cores and the cluster system at Department of Computing in Engineering with 24 processors.

The examples is used is Plane Truss Optimization with Master-Worker (MW) Paradigm.

First platform: Intel CPU with 4 multiple core processors.

Two examples of Plane Truss Optimization with Genetic Algorithm:

Example 1:

Generation: 500
Population: 100

Example 2:

Generation: 1000
Population: 100

<table>
<thead>
<tr>
<th>Method</th>
<th>Generation/Population</th>
<th>Number of workers</th>
<th>Time MPI (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWPipeline</td>
<td>500/100</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>MWPipeline</td>
<td>500/100</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>MWPipeline</td>
<td>500/100</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>MWPipeline</td>
<td>1000/100</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>MWPipeline</td>
<td>1000/100</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>MWPipeline</td>
<td>1000/100</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>MWUpdate</td>
<td>500/100</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>MWUpdate</td>
<td>500/100</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>MWUpdate</td>
<td>500/100</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>MWUpdate</td>
<td>1000/100</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>MWUpdate</td>
<td>1000/100</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>MWUpdate</td>
<td>1000/100</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5.3 Performance Statistic with Intel CPU 4 multiple cores
Figure 5.1 Result of speed up with Master-Worker in Pipeline method

Figure 5.2 Results of speed up with Master-Worker in Update Flow method with 3 new approaches

From the Figure 5.1 and Figure 5.2 it’s obviously that the increase in number of processors results to the decrease of the execution time, for example in Figure 5.1 in case with 500 generations and population is 100: from run time 13 seconds with 1 worker/2 processors down to 8 seconds with 2 workers/3 processors and then 7 seconds with 3 workers/4 processors respectively. The execution time was measured by the MPI run time.
Moreover, comparing between Figure 5.1 and Figure 5.2 shows that the impact in the increasing the speed up by applying the Update Flow with 3 new approach for the Master-Worker Paradigm.

Second platform: GPU Cluster at Department of Computing in Engineering with 24 processors

Test configuration:

Example 1: Plane Truss Optimization with Genetic Algorithm

Generation: 1000

Population: 100

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Number of Workers</th>
<th>Time MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>23</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.4 Performance Statistic with GPU Cluster run example 1
Figure 5.3 Results of the speed up run example 1 on GPU Cluster System

The Figure 5.3 shows the results of speed up when the example 1 is run on the GPU Cluster system. It not only shows that the increase of the number of processors results to the decrease of the run time but also it obviously points out that the speed up is not linear, it follows the Amdahl’s law.
Example 2: Plane Truss Optimization with Genetic Algorithm

Generation: 500

Population: 100

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Number of Workers</th>
<th>Time MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.5 Performance Statistic with GPU Cluster run example 2

The Figure 5.4 shows the results of speed up when the example 2 is run on the GPU Cluster system. Similarly to the Figure 5.3 it not only shows that the increase of the number of processors results to the decrease of the run time but also it obviously points out that the speed up is not linear, it follows the Amdahl’s law.
Figure 5. Results compare speed up between multiple cores with 1, 2, 3 workers and cluster with 23 workers.

The Figure 6.3 shows the difference of the speed up between two platforms: the computer with an Intel CPU 4 multiple cores and the cluster system at Department of Computing in Engineering. The reason of the difference due to many influenced factors such as the speed and effectiveness of the communication in cache mechanism of the multiple core compare with communication of the cluster system.
Chapter VI

Conclusion and Future Work

1. Conclusions
In the presented work, it has proposed a parallel method equipped with algorithms to solve a structural optimization, which emphasis on scalability and parallel implementation. The main objective was to design and implement algorithms and data structures for parallel with Master-Worker paradigm as a framework. Various different approaches and techniques for the Master-Worker Paradigm were illustrated in chapter 2 and also there are two main algorithms were chosen to implemented, that were described in chapter 3.

Presented algorithms were implemented in the prototype application and tested on several basic structural optimizations such as plane truss and space truss (and also a small basic example with Three Bar structure). The results indicate that the solution performs very well in parallel with multiple cores and it is reasonable to believe that it can utilize even far better performance when runs on cluster or deals with big structural optimization.

Finally, this document shows the dilemma of redundant programming, or there are obstacles with algorithm implementation for parallel, in case the parallel evaluation of predicates. Even though this concept seemed quite promising at the beginning, practical tests have shown that in common situations these techniques do not improve performance much and sometimes cannot perform as expected.

There are many questions still open for further research, however this research has successfully achieved the objectives and proved that it’s a good idea to build a framework for structural optimization using parallel computing.

2. Future Work
The first prototype framework implements only a subset structural optimization domain; however it would not be difficult for user to understand how the framework can be used for specified structural optimization problem as well as details of the implementation into his model. Also the framework still is not implemented with more algorithms, however, it carries the basic concepts, and hence it can be easily completed to full specification.

For future work, it is needed to implement more sophisticated algorithms to have more options and better performance for the proposed concept, and improve the master/worker
paradigm, e.g. a load balancing algorithm among master and workers. Development of these algorithms is to achieve a better performance in the future work.
Bibliography


[3] Asanovic and al. “Power is free, but transistors are expensive”.


Appendix

Appendix A: MPJ User Guide

1. Executing and Debugging MPJ Express Programs using the Eclipse IDE

Steps

1. Create a new Java project:

   a. Start Eclipse and go to File/New/Java Project. A window with the label “Create a Java Project” will appear asking for the project name (“MulticoreDebuggerDemo” in our case). Click on the next button.

   b. A new window appears with the label “Java Setting”, which allows adding external JAR files to the Eclipse project. Select the “Libraries” tab and click on the “Add External JARs…” button to add the mpj.jar file to the project. The mpj.jar file exists in the lib subdirectory of the MPJ Express software. Click the “Finish” button towards the bottom of the window. The new Java project “MulticoreDebuggerDemo” has been successfully created.

2. Write the source code:

   a. Go to File/NewClass and a window appears with the label “Java Class”. Write the class name “HelloEclipseWorld” in the “Name” textbox and press the “Finish” button. The HelloEclipseWorld.java class has been created.

   b. As a demo, write the following source code:

```java
import mpi.*;

public class HelloEclipseWorld {

 public static void main(String[] args) throws Exception {

     MPI.Init(args);

     int rank = MPI.COMM_WORLD.Rank();

     int size = MPI.COMM_WORLD.Size();

     System.out.println("I am process "+rank+" of total "+size+" processes.");

     MPI.Finalize();

 }

 ```

3. Execute the MPJ Express parallel program in the multicore mode:

   a. Goto the “Run” menu and select the “Run Configurations…” menu item. A window with the label “Create, manage, and run configurations” appears.

   b. Double click the “Java application” launch configuration, which is towards the left side of the window. After this a new launch configuration by the name of HelloEclipseWorld has been created that appears on the right side of the windows. This new window has several tabs including “Main”, “Arguments”, “JRE” etc. Select the “Arguments” tab and in the appeared window select the Arguments tab.

   c. To execute the MPJ Express HelloEclipseWorld program, we need to specify “-jar ${MPJ_HOME}/lib/starter.jar” arguments to the VM. But before specifying these arguments in the VM arguments textbox, first set the “MPJ_HOME” environment variable if it does not exist already.

   d. To specify the “MPJ_HOME” variable, click on the “Variables…” button below the VM arguments textbox. A window appears with the title “Select Variable”. If the MPJ_HOME variable exists, select it. Otherwise, click on the “Edit Variables…” button that triggers a new window. Click the “New” button on this window and set the value of “MPJ_HOME” variable to the root directory (for example D:\mpj-v0.36) of the MPJ Express software. Click the “OK” button when done.

   e. A window with the label “Create, manage, and run configurations” should be visible now. Click the “Run” button to execute the parallel MPJ Express program. Output of the program should appear in the console that typically appears near the bottom of the IDE. A user may change the total number of MPJ Express processes by using the switch “-np” in the VM arguments textbox. For example, to run the same program on four processing cores, add the “-np 4” to VM arguments.

4. Debug the MPJ Express parallel program in the multicore mode:

   a. Goto the “Run” menu and select the “Run Configurations…” menu item. A window with the label “Create, manage, and run configurations” appears. Select the “HelloEclipseWorld” launch configuration.

   b. Click the “Arguments” tab and add the string “-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8000” to the VM arguments textbox. The Eclipse debugger would attach itself with the port specified in this string—the value is 8000 by default and can be changed. Click on “Apply” button and press the “Close” button.
c. Open the “Debug” perspective of the Eclipse IDE and introduce breakpoints in the HelloEclipseWorld program.

d. Goto the “Run” menu and select the “Run Configurations…” menu item. A window with the label “Create, manage, and run configurations” appears. Click the “Run” button to execute the parallel program. The following output will appear on the console:

e. Goto the “Run” menu and select the “Debug Configurations…” menu item. A window with the label “Create, manage, and run configurations” appears. Double click “Remote Java Application” launch configuration to create the “HelloEclipseWorld (1)” instance - if it does not exist already. The default port for this instance is 8000 but can be changed if required. Click the “Debug” button.

f. The execution starts but all MPJ Express processes (or threads) hang their execution on the first breakpoint. Now the user can manage the execution of their program by selecting the “Debug” tab in the “Debug” perspective. Under the “HelloEclipseWorld (1) [Remote Java Application]” label, there will be an option corresponding to the main thread. In addition there will be other threads that are equal to the number of processes started by the user by specifying the “-np” switch. Execution of these threads can be controlled by using various debugging options provided by Eclipse IDE.