Material and Methods

3.1 Implementation Issue

The practical issues of Turbo code designing with simulation environment are discussed in this paper. The main topics include the Monte Carlo simulator. It is designed to allow multi-result experiments and it makes full use of multiprocessor facilities and the BCJR algorithm with associated numerical accuracy problems. Other topics that included in this chapter involve the provision of low-level facilities. The discussions in this chapter are very closer to the hardware and software implementation. All descriptions are kept at algorithm level design.

3.1.1 Software Language Choice

It is very flexibility if one simulator may be used to work with different communication designing. It is used to allow in different channel models, information sources, modulation schemes, source coding scheme and channel coding. In channel coding, the system may use an equiprobable binary source in coding, a direct mapping source code, BPSK modulation and AWGN channel. The every part needs to be replaceable in system and all this type of designing is done naturally with Object-Oriented Programming (OOP) model.

The second major requirement is a speed, suggests the use of an efficient compiled language. The further improvement in performance may be obtained by parallel execution of program at various levels of designed. Together with the Object-Oriented Programming (OOP) suggestion, the choice of the C++ programming language may be considered in coding scheme. In parallel programming, the Message Passing Interface model is selected for simplicity. This type of model performs the execution in parallel over a cluster of workstations in system. It is a very attractive for large grain parallelism type of system with smaller overheads in scheme.

3.2 Advantage of Monte Carlo Simulator

This simulator implementation differs from the basic system design, as described in Section below, in the following respects. Which is given below?

I. It gives the multiple results. For example, simulation of Turbo codes in communication system will return the BER and Frame Error Rate (FER) performance with iteration of coding.
The Monte Carlo simulator stops the searching when all results are obtained within tolerance limits of code.

II. It gives the opportunity of high efficiency utilization of resources and such parallelism is not communication intensive, facilitating the use of parallel systems with low interprocessor bandwidth, such as a multi-computer system.

3.2.1 Multi-Processing
In uniform sampling, the all samples are independent and it is simple to compute different samples on separate processing nodes in system. The independence results from different nodes are ensured by seeding the random generators differently for each node. In our implementation, we have used the Local Area Multicomputer library from Ohio Supercomputer Centre.

\begin{verbatim}
begin
  Initialize results
  Send a work command to all slaves
  repeat
    wait for any slave to return a result
    update estimate with slave’s result
    compute accuracy reached
    required accuracy not reached yet
      send a work command to slave that returned result
  end if
until required accuracy reached
  wait for results from remaining slaves
  update estimate with slave’s results
end

Root Process Algorithm
\end{verbatim}

3.2.2 Dynamic Load Balancing
The parallel system is implemented with Master-Slave model. The master (or root) process distributes the work among the slave processes, gathers the results, and computes the estimate.
The slaves are used to compute a sample and return its result. Since all samples are independent. The order of arrival of results is not important. This makes it very easy to implement dynamic load balancing for maximum utilization of resources in system. This type of basic algorithm is given in Fig. 3.2.3 Master-Slave Parallel System

To simplify the designing process, both master and slave component in the parallel system of programs are implemented in a single executable time. The parallel programs are started on different computer systems without shared memory and each slave must go through the initialization process separately. In this system, we must make sure that before parallel execution starts, all programs have the required data initialized with the same values. Similarly, a mechanism must be provided for the root process and change simulation parameters in the slaves. It simulates the same system at a different SNR. The initialization procedure is shown by the flowchart in Fig. 4.2. The program can check whether it is a master or slave by reading the size of the parent communicator. If it is zero size, then the running program in system must be the master, otherwise, it is a slave.

The root program must be initializing the whole system by checking the topology of the available system, and spawning a slave per processor. The slave routine has responded to the commands given by the master and exiting when this is requested. To avoid hogging resources, all slaves automatically decrease process of priority on initialization. It does not imply the performance penalty unless other users are also using the system. The cluster of workstations includes a number of single and dual-processor machines as well as a quad-processor. Due to limitations with MPI 1.1 standard regarding the initialization of processes, the standard LAM model only caters the machines as processing nodes, and processes are distributed in machines in round-robin fashion style and spawning has to be performed in a single MPI call. Because all slave processes in system need to be in the same MPI communicator. This makes it impossible to start exactly one process per processor in system.

Due to this problem, the class which handles a higher level interface to the LAM routines, written specifically for a client-server setup initially spawns a number of processes equal to the number of machines, and asks each process to return the number of processors in system on that particular machine. In this way, the root process has a table that containing the number of processors in each machine. The next, the root process spawns a number of processes equal
to N, The \( N \) is the number of machines and \( N \) is the number of processes per machine. Now while this is what we want for any machine with processors and there are extra processes on other machines. Thus, finally, the root process will kill \( n_i \) processes on each machine, \( n_i \) is the number of processors in machine \( i \). This type of algorithm is represented in Fig. 

### 3.2.4 Handling Dead-Lock

In certain conditions, a large proportion of the samples have a zero value. In this type of cases, the assumption of Gaussian profile is violated, and the estimate will be seen as follows

**Begin**

- Read the number of nodes (\( N \)) in the LAM system
- Spawn \( N \) processes (one per machine)
- For each of the \( N \) processes
  - Request process to return the number of CPU in that machine
  - Read the returned result and in a stable
  - Request process to die
- End for
- Find the largest no of processors per mode, \( v \)
- Spawn \( v \) \( N \) processer (\( v \) per machine)
- Kill extraneous processors on machines with less than \( v \) processors

**end**

### Root Process Initialization

Convergence is very slowly. This happens, for example, when computing the error-count profile. The solution is not to be consider the computed accuracy of such estimates when checking if the required accuracy has been reached. This is done by considering only results for which the fraction of non-zero estimates is greater than one.
3.3 BCJR Algorithm

A major implementation of the probability-based BCJR algorithm is the large numerical range of the metrics; it increases exponentially with block size in system. This problem has led
researchers around the world to implement the BCJR algorithm in the Log domain (LLR). Our solution is to implement the BCJR algorithm by using the template feature of C++ to allow the use of any mathematical precision required. A number of higher precision arithmetic classes are used in our implementation process and providing a range of compromises between accuracy and speed, memory requirements. These terms are described as below an evaluation of their accuracy and speed, and how well they perform within a Turbo code system. It is considered in the following chapter.[30]

3.3.1 The Infinite Precision
Initially, the accuracy problem was avoided by providing a class which can compute on real numbers with infinite accuracy. The class is built as a C++ interface to the Multi-Precision library by using overloaded operators for addition, subtraction, multiplication and division. Conversion to and from double-precision values is also provided. While its usefulness is undeniable, such an implementation naturally suffers a significant performance. It is disadvantage.

3.3.2 Extended Precision Floating Point Computation
Since the BCJR algorithm requirement is extended range rather than increased precision. The optimized solution of the problem is provided by a class with a 32-bit exponent. To simplify implementation, the mantissa is held as a standard IEEE 64-bit floating-point number in the system. After each arithmetic operation, the result is normalized. This type of class results in a significant increase in performance and reduction in memory space is obtained in the coding system. However, both speed and memory are still compromised for the benefit of accuracy in system.

3.3.3 Log-Scale Floating-Point Computation
The usual solution of log scale floating point in practical Turbo codes is to store the natural logarithms of the numbers, rather than the value itself. While this clearly solves the range problem, its effect on the resolution of the numbers will be not clear. Working with logarithms will also introduce a new difficulty such as computing the addition of two numbers. A practical solution is to compute the addition of A and B using the corresponding Logarithm values a, b as follows:
\[
\log(A+B) = \log(e^a + e^b) = \min(a, b) - \log(1 + e^{a-b})
\]

The term \(a = \log(A)\) and \(b = \log(B)\). Thus, the addition becomes a comparison operation (select the smaller value \(a\) or \(b\)) and a function that depends on the difference between \(a\) and \(b\).

It can be shown that the largest value of \(f(c)\) is equal to \(\log(2)\) and that as \(c\) increases, \(f(c)\) quickly tends to zero. This means that \(f(c)\) can be easily implemented by using a small lookup table with quantizing \(c\).

### 3.4 Supporting Facilities

#### 3.4.1 Random Number Generator

The Pseudo-random sequences are used in various portion of the system. It will be used in channel model and a number of interleaver creation techniques. The random generator must provide the following interface. Which is given below?

1. Seeding function.
2. A function which returns as random integer modulo \(m\), where \(m\) is user parameter and a function returning a random floating point number in \([0, 1]\). The numbers returned by both of these functions in this type of coding have a flat probability distribution in the allowed range of sequence.
3. Another function that returns the random floating-point number with a Gaussian probability distribution with zero mean and unit variance, or user supplied variance. Random sequences are generated using the subtractive algorithm. Our implementation is based on the description given by Press et al. [1992]. This type of algorithm was chosen because it has a very long period and it does not suffer from low order correlations. It simply generates the random bit sequences.

The Gaussian profile reshaping is performed by using the Box-Muller Transform [Press et al., 1992]. This transform has the advantage that, two Gaussian random numbers are generated with help of two random numbers with a flat profile. Thus, the period of the underlying generator is not reduced. Another good property is that the profile of the random values generated in system is exactly Gaussian order.

### 3.5 Testing and simulation environment.
In this chapter we introduce the testing method involved the validity of the simpler modules. Those can be tested separately and it proceeding the more complex modules. Which make use of other modules internally in system? In presence, we first tested the Monte Carlo simulator with an uncoded transmission in system. This is because the expected BER can be easily computed mathematically for comparison. Next, we test the MAP decoder by comparing the simulated results with a Union bound for BER. We also confirm the accuracy of the various numerical representations in system that is used in BCJR algorithm by comparing simulation results for the different techniques. Finally, we test the Turbo decoder by comparing with known bounds results and with published results.

3.5.1 Uncoded Wireless Transmission
The initial testing involved the uncoded transmission of information using BPSK modulation over the AWGN channel. The expected bit error rate can be easily calculated for comparison with the simulation results [Proakis, 1995]:

\[ P_b = Q \]  

3.3
The blocks of 1000 bits were transmitted without channel coding and an estimate of the BER is computed by using the Monte Carlo algorithm with a tolerance of ±10% at a confidence level of 95%. This is compared with the expected BER with help of Eqn. (3.3). It is clear that the expected value is well within the simulation tolerance limits. It is also shown in the figure.

### 3.5.2 The Upper Bound for MAP Decoder

The bit error performance of a linear block code may be bounded by the union bound condition. It is represented by Cover & Thomas, 1991.

\[ P_b(w, d) \leq Q(3.4) \]
In the equation, the terms $A(w, d)$ is represented by number of codeword of input weight $w$ and total weight $d$. The code’s block size is represented by the number of information bits $\tau$ and the code rate $R$ in coding system.

![BER Performance of Union Bound](image)

**Fig 3.3: BER Performance of Union Bound**

We consider the rate 1/2 RSC code. Now, encoding an information block of 24 bits with 2 bits for tailing bit will result in a (52; 24) equivalent block code. After obtaining the $A(w, d)$ matrix up to $d = 20$ for this type of code, the union bound can be computed. The code’s performance is simulated by using a MAP decoder. This is based on the BCJR algorithm. Fig. 3.3 compares the simulation result with the union bound. It is shown in figure for increasing codeword weight with considering in the summation result. It is clear that, the bound is very tight for high SNR and bound converges rapidly in this system.

### 3.5.3 BCJR Arithmetic Accuracy

In order to confirm the accuracy of the faster arithmetic routines used to speed up the BCJR algorithm. It is discussed in previous Section. We simulate the same terminated RSC code by using all three arithmetic models in system.
It is seen in Fig. 3.4, the faster routines does not result in any noticeable difference in the performance of coding.

### 3.5.4 Upper Bound for Turbo Decoder

The union bound of Eqn. (3.4) can be used with Turbo codes. We can obtain the $A(w, d)$ matrix. In Turbo codes, this is not a very trivial task because the block size is usually large and the presence of the interleaver complicates the problem in coding scheme. There are two solutions to this problem; the first is to work with very small block sizes. The second solution is that it can be used with the larger block sizes normally associated with Turbo codes with presence of a uniform interleaver. Turbo code with rate $1/3$ is created by using RSC component codes with generator $(1, 7/5)$ and a uniform interleaver. We assume that information block length is 1000 bits and both the original and the interleaved sequences are tailed by using the method described by Divsalar and Pollara [1995].
Table 3.2: Coefficients used to compute BER bound for (3006; 1000) Turbo code

<table>
<thead>
<tr>
<th>S No</th>
<th>d</th>
<th>( A_d )</th>
<th>S No</th>
<th>d</th>
<th>( A_d )</th>
<th>S No</th>
<th>d</th>
<th>( A_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>0.0039821</td>
<td>10</td>
<td>17</td>
<td>0.1457</td>
<td>19</td>
<td>26</td>
<td>10.67</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>0.0079605</td>
<td>11</td>
<td>18</td>
<td>0.2984</td>
<td>20</td>
<td>27</td>
<td>17.65</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0.011918</td>
<td>12</td>
<td>19</td>
<td>0.5472</td>
<td>21</td>
<td>28</td>
<td>29.61</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>0.015861</td>
<td>13</td>
<td>20</td>
<td>0.9171</td>
<td>22</td>
<td>29</td>
<td>49.31</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>0.019887</td>
<td>14</td>
<td>21</td>
<td>1.437</td>
<td>23</td>
<td>30</td>
<td>80.57</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>0.024188</td>
<td>15</td>
<td>22</td>
<td>2.144</td>
<td>24</td>
<td>31</td>
<td>128.6</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>0.029048</td>
<td>16</td>
<td>23</td>
<td>3.09</td>
<td>25</td>
<td>32</td>
<td>201.3</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>0.034846</td>
<td>17</td>
<td>24</td>
<td>4.465</td>
<td>26</td>
<td>33</td>
<td>311.5</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>0.065768</td>
<td>18</td>
<td>25</td>
<td>6.716</td>
<td>27</td>
<td>34</td>
<td>481.2</td>
</tr>
</tbody>
</table>

Figure 3.5: Turbo decoder compared with Union Bound

The results of (3006; 1000) Turbo code is shown in table. The union bound can be computed by using the following equation.

\[
P_b = d \cdot Q
\]

3.5
The $A_d$ is indicated by total information weight in all codeword of weight $d$ divided by number of information bits per codeword in equation. It is defined by Eqn. The summation usually operates on a truncated set of codeword weights in turbo code. In the Turbo code with $\tau = 1000$, the set of coefficients used to compute the bound is given in Table. The union bound result is compared with a simulation of the code’s performance in Fig.

### 3.5.5 Comparison with Published Results

![Figure 3.6: BIT error rate performance of iteration (208, 102)](image)

We have compared our simulator results with published by other independent researchers. The first setup that we replicated consists of a rate 1/2 Turbo code; it is constructed from two RSC component codes with generator $(1; 5/7)$, a helical interleaver with 17 rows and 6 columns and an odd-even puncturing pattern. The 102-bit information sequence is passed through the interleaver with a 2-bit tail appended. It is represented by a $(208; 102)$ Turbo code. Since the interleaver is similar and the same tail can be used for both interleaved and non-interleaved sequences. Our simulation results are shown in Figure below, agree with those published by Barbulescu [1996, p. 41] for 8 iterations (our code is the same as the TNIE code given in Fig. 3.7). The second setup is for a rate 1/3 Turbo code constructed from two RSC component codes with generator $(1; 5/7)$ and a 1024-bit random interleaver.
Turbo code terminated by JPL scheme is shown in figure. Our simulation results after ten iterations are shown in Fig 3.7, together with the results quoted by Montorsi. Note that for all points, we simulated more than twice the number of blocks used by Montorsi.

![Figure 3.7: Turbo decoder comparison with Montorsi](image)

Furthermore, our choice for the number of blocks to be simulated is not ad hoc, but is based on the required tolerance level and confidence level. We also observed that for a small number of samples (less than 100), the distribution of sample results is skewed.

<table>
<thead>
<tr>
<th>Table 3:3 Turbo decoder comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>S No</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>