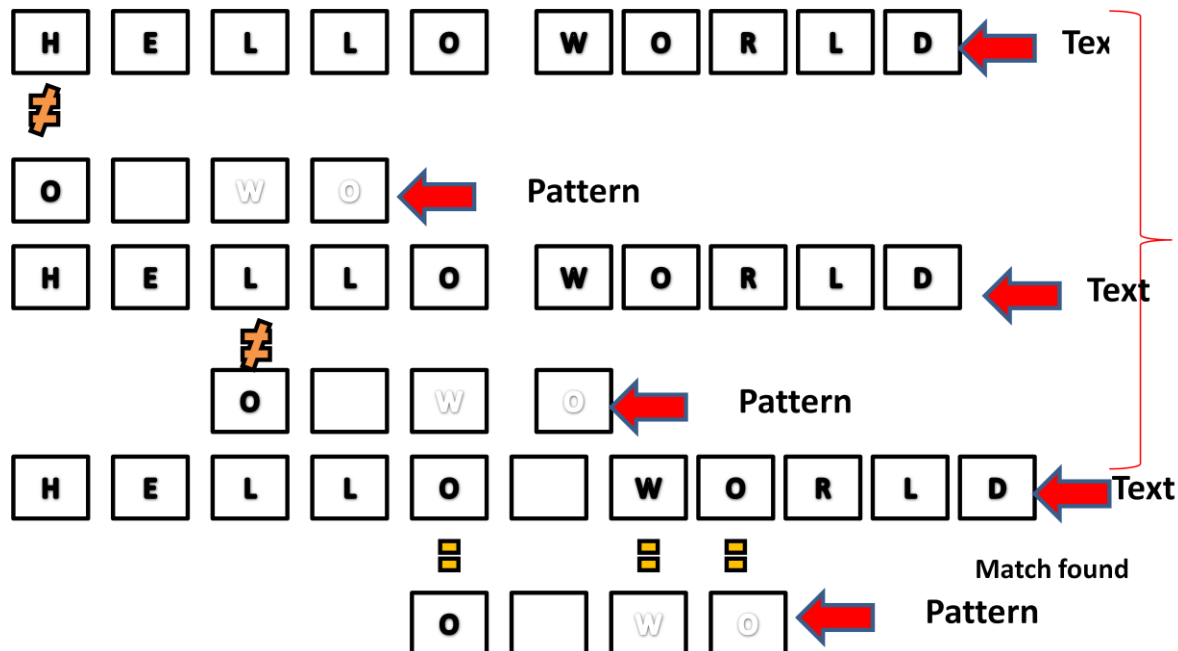


## 1 .INTRODUCTION

### String Matching:

- String searching algorithms, sometimes called string matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. Most basic approach employed in string matching would be Brute Force Algorithm<sup>[1]</sup>.

Example



### Boyer-Moore-Horspool Algorithm:

Boyer-Moore-Horspool algorithm or Horspool's algorithm is an algorithm for finding substrings in strings. It was published by Nigel Horspool in 1980. It is a simplification of the Boyer-Moore string search algorithm which is related to the Knuth-Morris-Pratt algorithm<sup>[1]</sup>.

<sup>[5]</sup>The Boyer-Moore algorithm worked on the principal that if we know the character in  $t[i+m-1]$  is not contained in the pattern, then we can shift over this position to  $i=i+m$ . Horspool suggested the use of shifting based on the last compared letter in the text, to match the last occurrence of that letter to the left of  $p[m-1]$ . This has a worst case run time of  $O(mn)$ .

### **Bad Character Rule:**

The bad-character rule considers the character in T at which the comparison process failed (assuming such a failure occurred). The next occurrence of that character to the left in P is found, and a shift which brings that occurrence in line with the mismatched occurrence in T is proposed. If the mismatched character does not occur to the left in P, a shift is proposed that moves the entirety of P past the point of mismatch.

### **Preprocessing :**

Methods vary on the exact form the table for the bad character rule should take, but a simple constant-time lookup solution is as follows: create a 2D table which is indexed first by the index of the character c in the alphabet and second by the index i in the pattern. This lookup will return the occurrence of c in P with the next-highest index  $j < i$  or -1 if there is no such occurrence. The proposed shift will then be  $i - j$ , with  $O(1)$  lookup time and  $O(kn)$  space, assuming a finite alphabet of length k.

### **Searching:**

The searching phase has a quadratic worst case but it can be proved that the average number of comparisons for one text character is between  $1/$  and  $2/(+1)$ .

### **Example:**

#### **Preprocessing:**

<sup>[2]</sup>A table is constructed by going through the pattern and process them based on their occurrence. The last character is not taken into consideration.

Here our pattern is "GCAGAGAG". The pattern is navigated from the 1st index till length – 2 characters. Based on Bad Character Rule a table with the character as index with corresponding value is created. While searching this table is looked into and accordingly the jumps are made to the next characters of the text.

<i>a</i>	A	C	G	T
<i>bmBc[a]</i>	1	6	2	8

**Searching:**

**First attempt**

G C A T C G C A G A G A G T A T A C A G T A C G  
1  
G C A G A G A G

Shift by: 1 ( $bmBc[A]$ )

**Second attempt**

G C A T C G C A G A G A G T A T A C A G T A C G  
2 1  
G C A G A G A G

Shift by: 2 ( $bmBc[G]$ )

**Third attempt**

G C A T C G C A G A G A G T A T A C A G T A C G  
2 1  
G C A G A G A G

Shift by: 2 ( $bmBc[G]$ )

**Fourth attempt**

G C A T C G C A G A G A G T A T A C A G T A C G  
2 3 4 5 6 7 8 1  
G C A G A G A G **MATCH**

## 2 .SEQUENTIAL CODE

### a) Source code written in C++

The sequential code has been built on reference code obtained in this link <sup>[3]</sup>

In this report we will highlight and explain important sections of the code .The full code has been submitted along with this report for complete reference.

```
typedef vector<size_t> occtable_type; // table type for occurrence table

const occtable_type create_table(const unsigned char* str , size_t str_length )
{
    occtable_type occ(UCHAR_MAX+1,str_length);

    if(str_length >= 1)
    {
        for( size_t i=0; i<(str_length-1); ++i)
            occ[str[i]] = (str_length -1)-i;
    }

    return occ;
}
```

The above function create\_table is used to create the occurrence table which holds values corresponding to each character in pattern. This value determines how much the pattern should be moved in order to continue the matching. An occurrence table generated by this code looks like :

The occurrence table Generated is :	
t	7
h	6
e	5
	4
k	3
i	2
n	1
g	8

The values on the right indicate the no. of characters that pattern has to be shifted. For eg, if e is the character matched , we has to move the pattern 5 places.

```
ifstream in;
in.open(filename);
while (!in.eof())
{
    //getting one line in the file
    getline(in,temp_str);

    //Converting the line into lower case to avoid case sensitive search
    transform(temp_str.begin(),temp_str.end(),temp_str.begin(), ::tolower);

    //Appending line to final string of txt file
    file_str.append(temp_str);
}
in.close();
```

This piece of code takes a .txt file specified in filename and inputs its content into a string for processing. **An assumption we make here is , all the string searches are not going to be case sensitive.** Hence , we convert the line to lower case before adding it to the sting.

```
while(text_pos <= (text_len - str_len ))
{
    // Selecting character at position equal to pattern length . -1 is to nullify
    // the 0th element.
    occ_char = text[text_pos + str_len -1];

    // If last character of pattern matches current character in text and
    // if characters ahead of current in text contains pattern , we have a match.
    // memcmp compares characters ahead of current with the pattern in blocks of
    // str_len -1.

    if (occ_char == str[str_len-1] && (memcmp(str, text+text_pos, str_len - 1) == 0))
    {
        count++;
    }

    // Look at occurrence table and decide how to increment text pointer
    text_pos += occ1[occ_char];
}
```

This is the most important block of code. We match last character of pattern to current text pointer . If the character is same , memcmp compares the characters ahead of current text pointer in blocks of size equal to pattern size. If memcmp returns a 0 , it means it found a set of characters equal to pattern . Hence there is a match and we update count. Then we look at occurrence table and update the text pointer position accordingly.

## **b) Alternate C++ implementation**

This source code has been built by referring the link <sup>[4]</sup>

This version is the classic version of the Boyer-Moore-Horsepool algorithm, wherein like the previous code, a 2-D table is created as a part of pre-processing. But the searching here occurs from right – left from the last character.

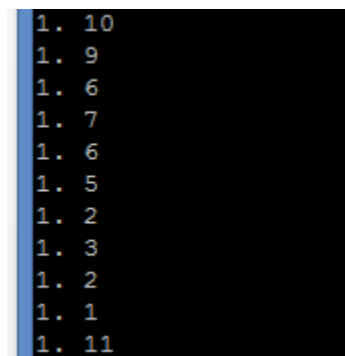
The values indexed by the characters in the pattern follow the logic as given in the following piece of code.

```
/* ---- Preprocess & create a table ---- */
for (scan = 0; scan <= UCHAR_MAX; scan = scan + 1)
    bad_char_skip[scan] = nlen;

/* C arrays have the first byte at [0], therefore:
 * [nlen - 1] is the last byte of the array. */
size_t last = nlen - 1;

/* Then populate it with the analysis of the needle */
for (scan = 0; scan < last; scan = scan + 1)
{
    bad_char_skip[needle[scan]] = last - scan;
}
}
```

While searching through the main text string, whenever there is partial or complete mismatch, the table created above is referenced in order to find the number of bits, the pattern has to shift. Thus we minimize the number of comparisons here, thus reducing the time complexity



```
1. 10
1. 9
1. 6
1. 7
1. 6
1. 5
1. 2
1. 3
1. 2
1. 1
1. 11
```

Above values are generated for the pattern “the economy” while pre-processing. When a pattern encounters any letter that isn’t matching the last letter, it looks up at the table and jumps accordingly.

Searching is slightly different from the previous version of the source code. Here firstly we enter the while loop by making comparisons with the haystack (text) and the needle length (pattern).

Here we scan the string with the last character of the pattern. If there is a match we move left till the beginning character of the pattern. Thus we loop within the “for” loop until we have matched till the beginning character. At this point we enter the “if” loop and here we have our entire pattern matched once.

```
/* ---- Do the matching ---- */
while (hlen >= nlen)
{
    /* scan from the end of the needle */
    for (scan = last; haystack[scan] == needle[scan]; scan = scan - 1)
    {
        if (scan == 0)
        { /* If the first byte matches, we've found it. */
            count++;
        }
    }
    hlen -= bad_char_skip[haystack[last]];
    haystack += bad_char_skip[haystack[last]];
}
cout << endl << " Number of times, string matched with the text : " << count << endl;
}
```

If there is a mismatch, it fails the “for” loop and looks up at the table. The pattern accordingly shifts based on the character values in the table created in the pre-processing phase. Thus here the mismatched characters are evicted out of the text at every mismatch. Also the “hlen” (text length) after every match is decreased by the number of bits jumped.

### **c) MATLAB sequential implementation**

- In the file stringmatch\_seq.m, we have created individual files for the functions and the whole assembly is run by running the main file.
- The sequential implementation in MATLAB is very much similar to the C++ implementation above just that we created individual files for each function.

### **d) Hardware targeted:**

We tested both our sequential versions of code in the discovery cluster[5] with the following specifications.

Cluster : Discovery Cluster

Processor : Intel Xeon E5 - 2670

No. of cores per CPU : 8

No. of hardware threads : 16

Clock Speed : 2.6 Ghz

Cache: 20 MB

Generally string search algorithms aren't computationally intensive so the above hardware was more than enough to test our sequential codes.

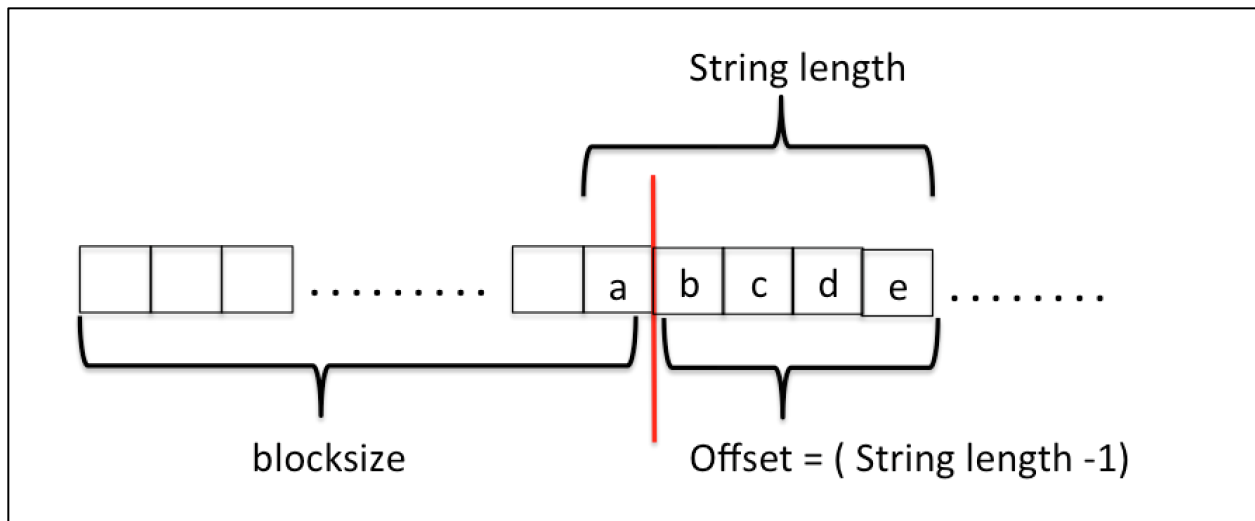
#### d) Timing:

We took the overall execution time after submitting the code in batch to the discovery cluster. The timing is noted from the output file generated in each implementation.

### 3. PARRALELIZATION IMPLEMENTED

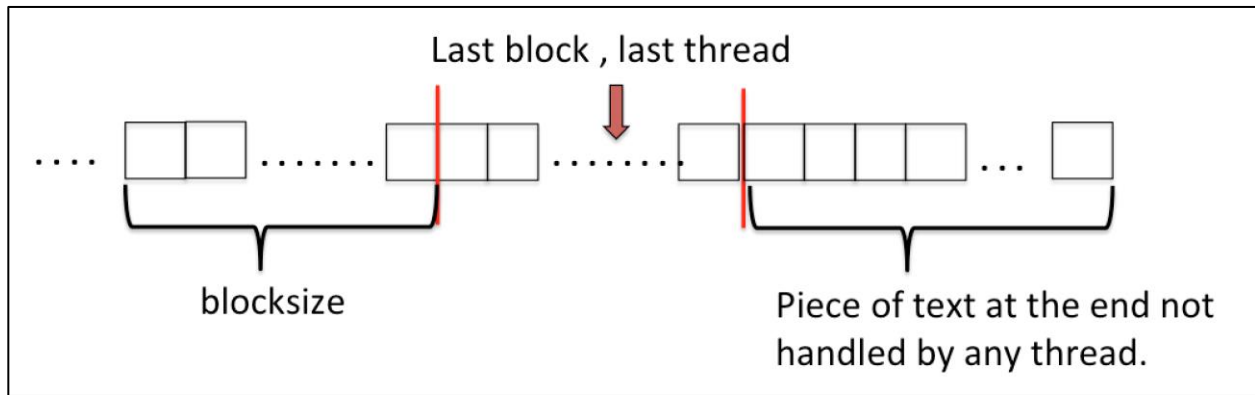
#### a. Open MP:

- We have employed the **partitioning strategy** of parallelism by dividing the text into small blocks based on the number of threads.
- Here each thread handles a block of text and processes it. Indexing the thread by  $(\text{blocksize} * \text{tid})$  makes sure that no thread picks the same block picked by the other thread.
- In order to avoid the case where text is cut in middle of the string , we set an offset to each block which is of value  $(\text{str\_len} - 1)$ .



- We have mapped the last thread such that it takes care of the reminder of the text which is not initially assigned to any of the threads.





- We didn't have to completely change the code since we had to parallelize only the part where actual comparisons take place.
- Other additions to the code included mapping of threads and categorizing the variables into private and shared.

### **Issues faced:**

- Firstly, we were trying to implement Open MP on the traditional version of the Horspool algorithm. But owing to lot of bank conflicts it became really difficult to efficiently map the threads.
- We also faced issues coming up with the values of offset. The number of string matches comes close to the actual count but it isn't accurate. This was solved by applying an offset of String length -1 as shown in the above picture so that when we iterate through the text we also consider offset positions.

### **b. MPI:**

- The approach used for MPI is similar to OpenMP ,except the procedure for passing blocks of text to each process.
- We use the command `MPI_File_Open` which basically opens the file on all processes .
- Then we use `MPI_File_read_at_all` by reading a block between defined start and end points as follows :

$\text{start} = \text{rank} * \text{blocksize}$

$\text{end} = ( \text{start} + \text{blocksize} + \text{offset} - 1 )$

- We also incorporate considerations end of text and offset.
- Each block is passed to the while loop there on , similar to sequential processing

### **Issues faced**

- We used MPI\_Scatter and MPI\_Gather to pass blocks of data to each process. However, this did not seem to work for our scenario, hence we decided to use the alternate approach of MPI\_File\_Open.

### **Hardware targeted:**

The same hardware environment as used for the sequential code was used for testing the OpenMP implementation.

### **c. Combination of Open MP and MPI:**

- The approach followed here is to implement Open MP code in each process generated using MPI.
- The block of text being generated by MPI is then processed using openMP by further dividing the block into no. of threads launched.
- The results of each process are added at the end just as in MPI.

### **Issues faced**

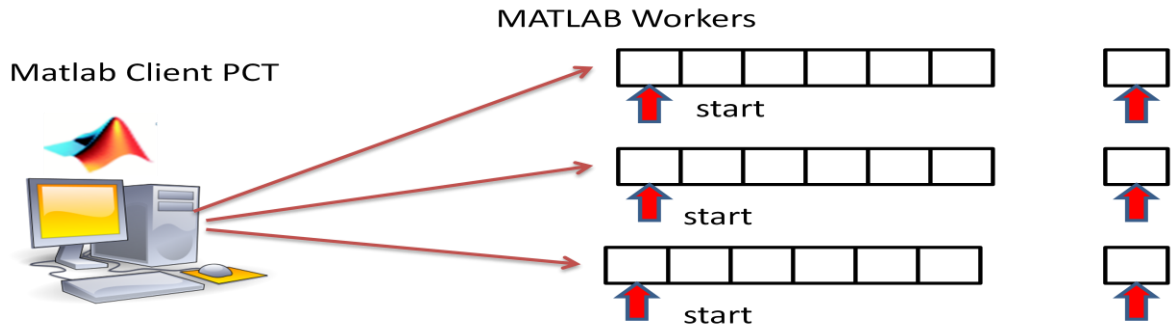
Since most of the issues were tackled in the individual OpenMP and MPI implementations, we did not face much issues here.

### **Hardware targeted:**

The same hardware environment as used for the sequential code was used for testing the OpenMP implementation.

### **d. MATLAB:**

- The approach followed in MATLAB is similar to MPI, wherein we create a job and subsequently create multiple tasks in this job.
- Each task has a *startValue* and *endValue* defined for it. These values determine the block of text that will be processed by the task.
- The job is submitted to the cluster and we wait for all the independent tasks to complete.
- Once the tasks are complete we calculate the sum of individual hits of the string found in each of the tasks.



```
createTask(job, @boyer_horsepool, 1, {text, string, text_len, str_len, start[index], end[index]}
```

### Issues faced

- We faced issues with the configurations of the COE cluster for MATLAB. We took the help of Nilay Roy to fix these.
- When creating tasks, we had issues in the function that is defined in file 'boyer.m'. The issue was the loop inside the code was running infinitely which was fixed by correcting the loop parameters.

### Hardware targeted:

Each nodes : 2 CPUs

Each CPU : 8 cores (total 16 cores)

We used 16 cores for splitting the tasks. No of tasks was varied between 2 and 16.

## 4. RESULTS

File : alice.txt

Length = 186792 characters

### Sequential Implementations :

Pattern	No.of occurrences	C ++ (seconds)	MATLAB (seconds)
alice	406	0.05	1.47

Parallel Implementations :

a. Open MP

No of threads	Time (seconds)
2	0.20
8	0.05
16	0.13
32	0.23

b. MPI

No of processes	Time (seconds)
1	0.22
2	0.36
4	1.21

c. OpenMP MPI combination

No of processes	No. of threads	Time (seconds)
1	2	0.23
	8	0.25
	16	0.54
2	2	0.31
	8	0.57
	16	1.34
4	2	0.57
	8	2.25
	16	3.7

d. MATLAB

Processes per node	Tasks	Time (seconds)
16	2.00	22.99
	4.00	23.70
	8.00	43.02
8	2	22.35
	4.00	23.53
	8.00	44.05
2	2.00	22.12
	4.00	25.22
	8	45.95

File : world192.txt

Length = 3040777 characters

Sequential Implementations :

Pattern	No.of occurrences	C ++ (seconds)	MATLAB (seconds)
and this string is added to help in better analysis	13177	0.19	180.4

Parallel Implementations :

a. Open MP

No of threads	Time (seconds)
2	0.3
8	0.36
16	0.56
32	0.89

b. MPI

No of processes	Time (seconds)
1	0.22
2	0.3
4	0.62

c. OpenMP MPI combination

No of processes	No. of threads	Time (seconds)
1	2	0.24
	8	0.38
	16	0.68
2	2	0.35
	8	0.75
	16	1.64
4	2	0.52
	8	2.27
	16	3.84

d. MATLAB

Processes per node	Tasks	Time (seconds)
16.00	2	148.30
	4	183.6
	8	400
8	2	262.84
	4	210.00
	8	289.55
2.00	2.	110.28
	4	69.60
	8	166.84

File : Gene.txt Length = 70408095 characters

Sequential Implementations :

Pattern	No.of occurrences	C ++ (seconds)	MATLAB (seconds)
Gttggta	335	0.37	201

Parallel Implementations :

a. Open MP

No of threads	Time (seconds)
2	0.589
8	0.93
16	2.605
32	2.82

b. MPI

No of processes	Time (seconds)
1	0.23
2	0.3
4	0.52

c. OpenMP MPI combination

No of process	No. of threads	Time (seconds)
1	2	0.24
	8	0.34
	16	0.77
		.
2	2	0.36
	8	0.65
	16	1.28
4	2	0.74
	8	2.28
	16	3.63

d. MATLAB

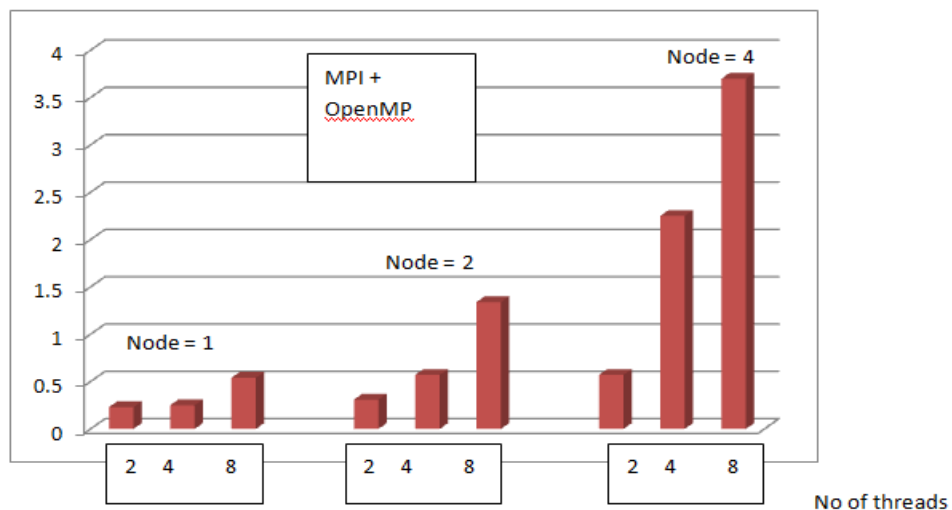
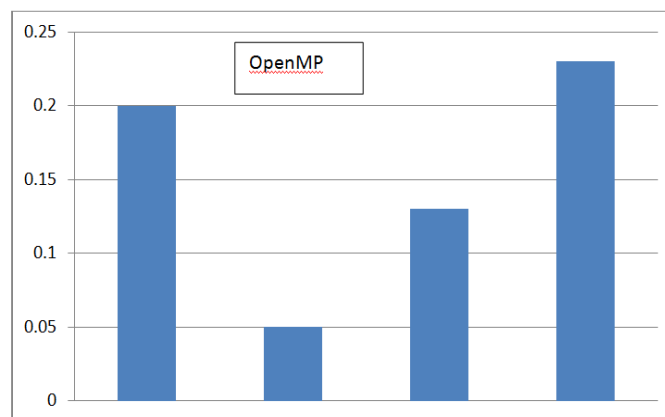
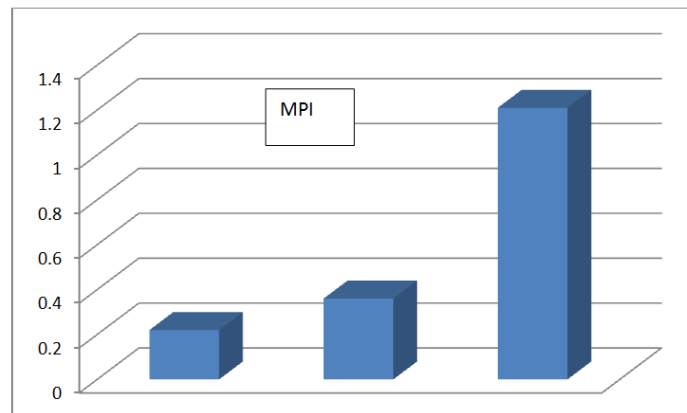
Processes per node	Tasks	Time (seconds)
16	2	149.45
	4	170.33
	8	329.62
8	2	271.99
	4	220.12
	8	280.11
2	2.	120.45
	4	172.44
	8	350.88

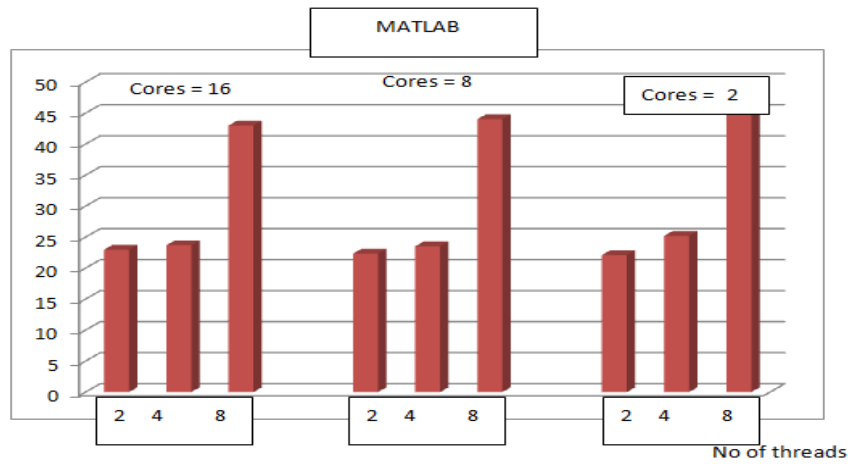
**Graphs:**

- The graphs are firstly plotted for respective sequential and parallel methods for a given text file. They are represented by bar graphs.
- Best timing of sequential, openmp, MPI and MPI + openmp is taken, compared and plotted. MATLAB implementation is not taken because the readings were vastly different and comparing with MATLAB sequential made more sense. The comparison graphs are plotted in pie chart.

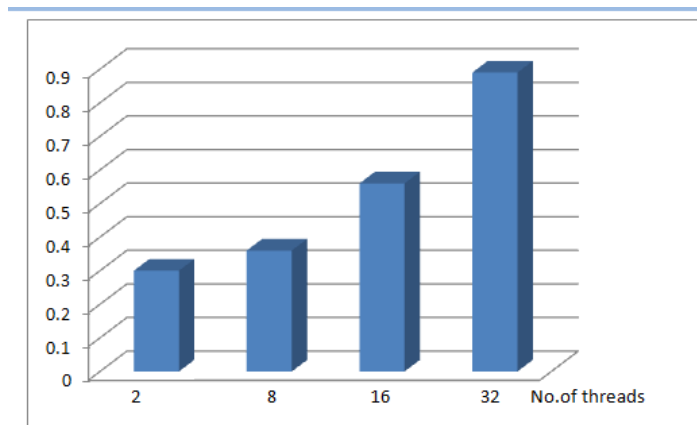


Alice.txt



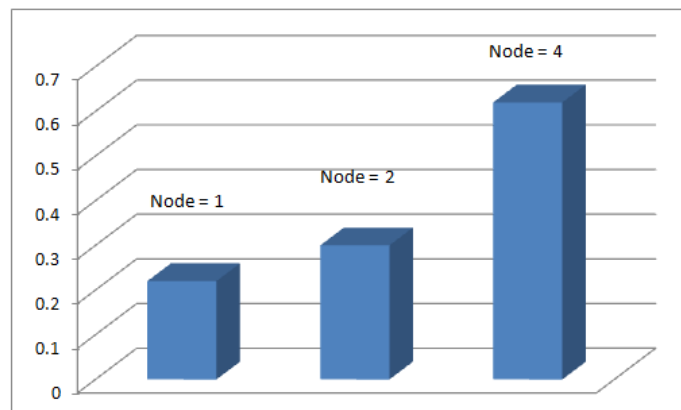


World192.txt

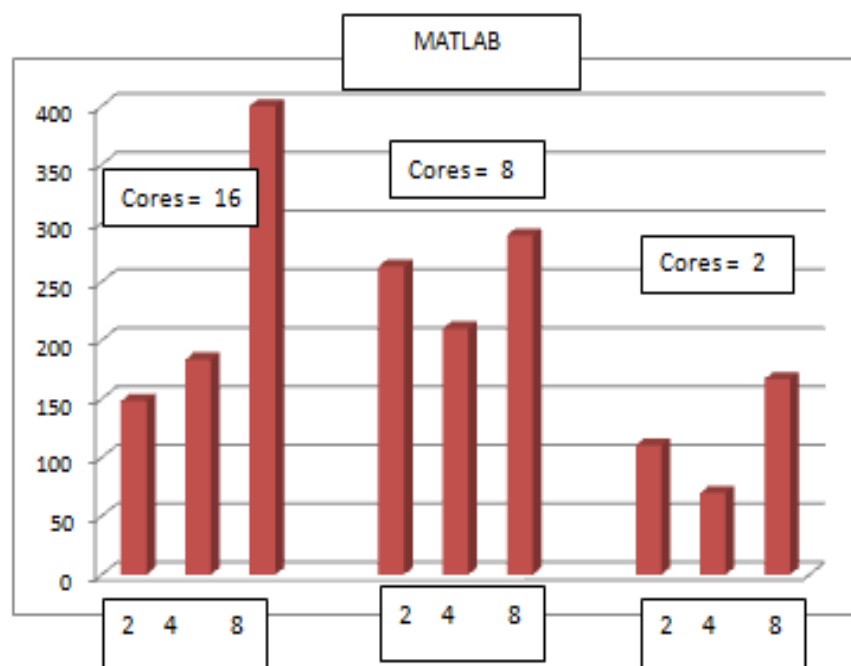
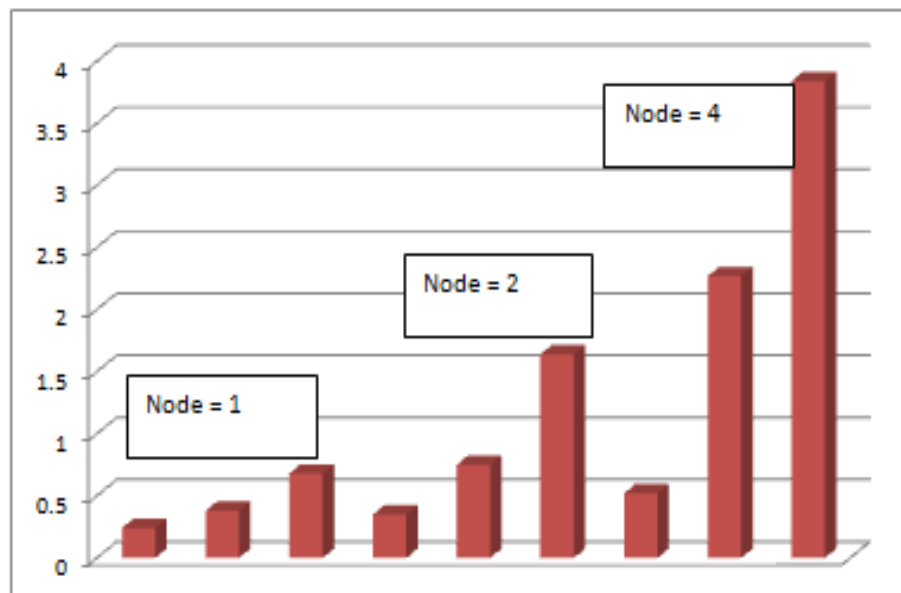


OpenMP

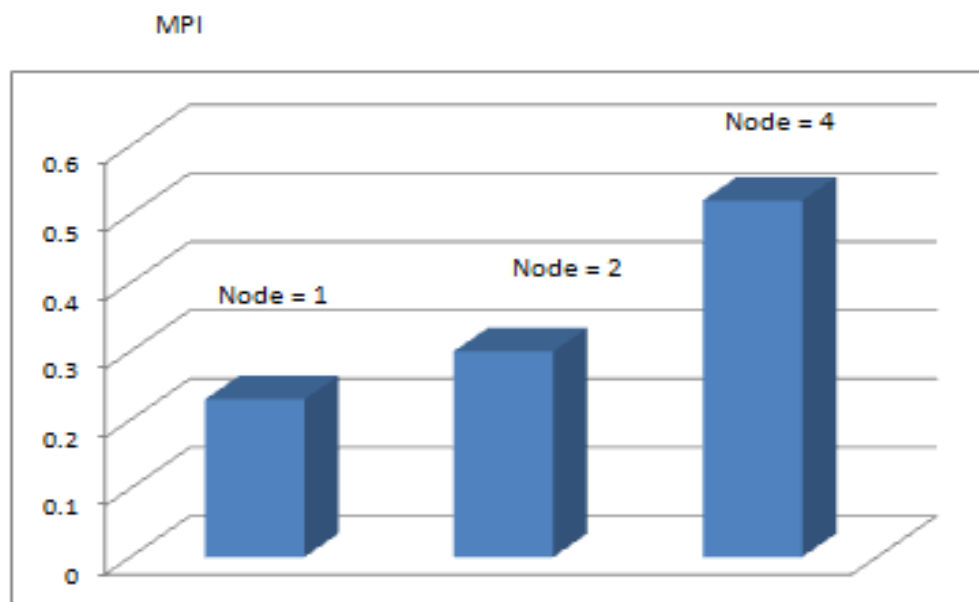
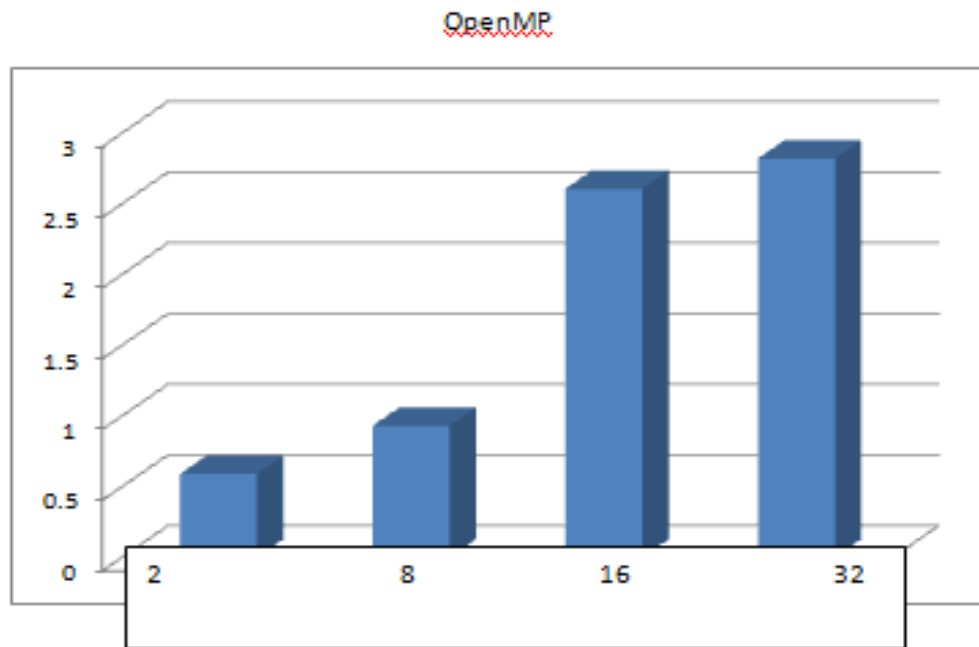
MPI



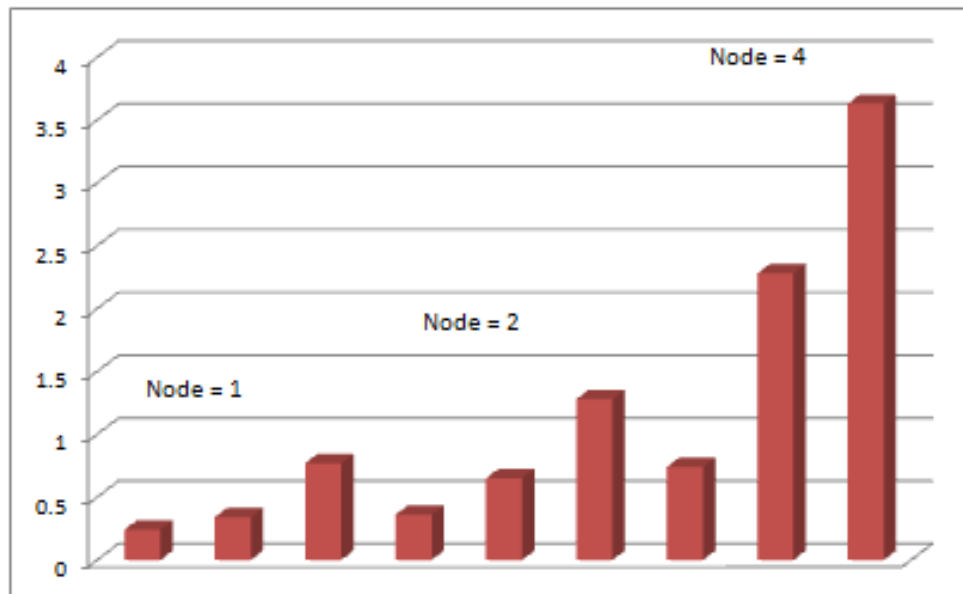
OpenMP + MPI



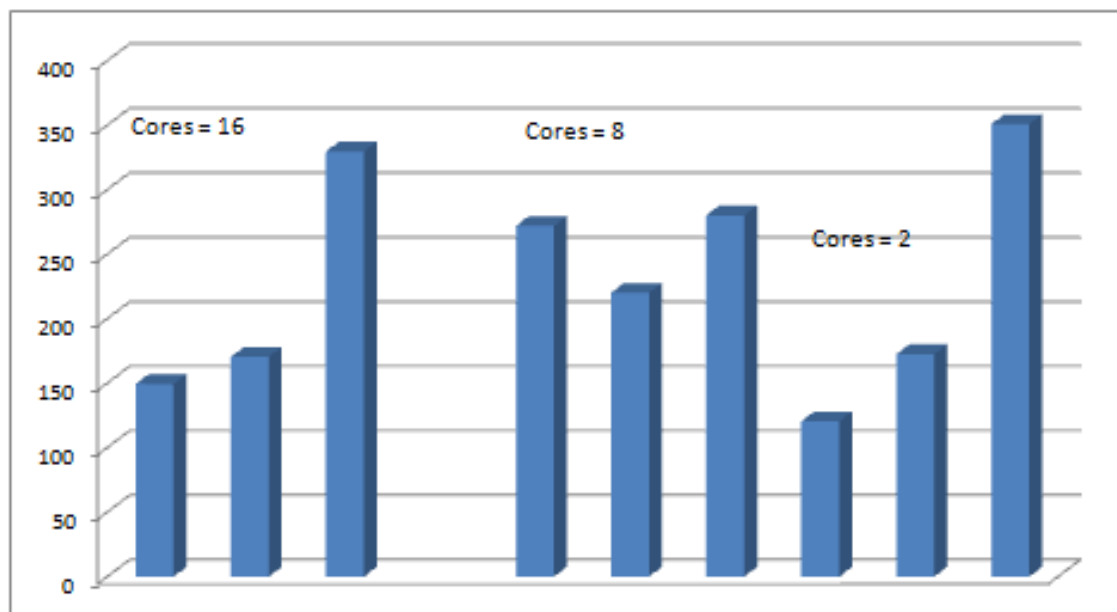
Gene.txt



MPI + OpenMP

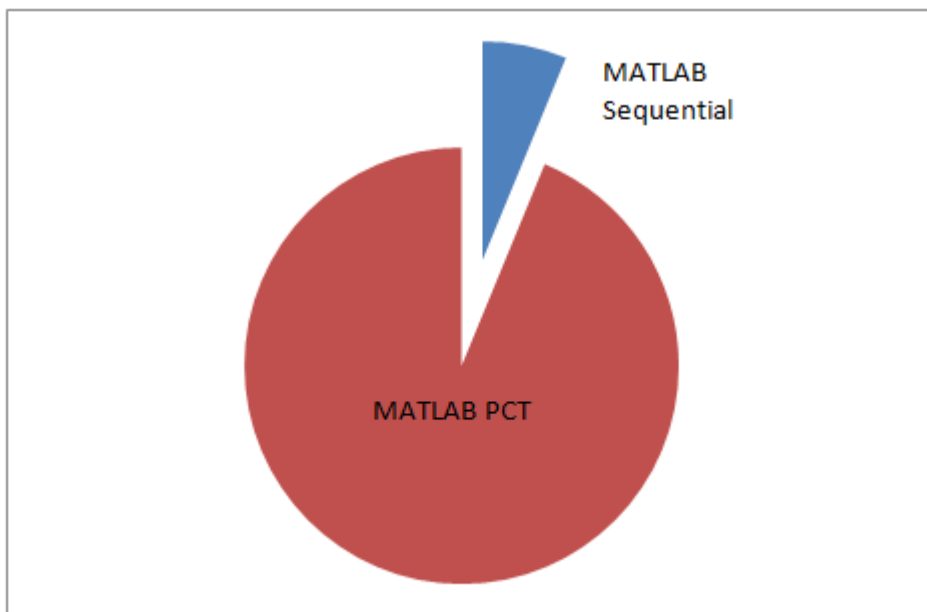
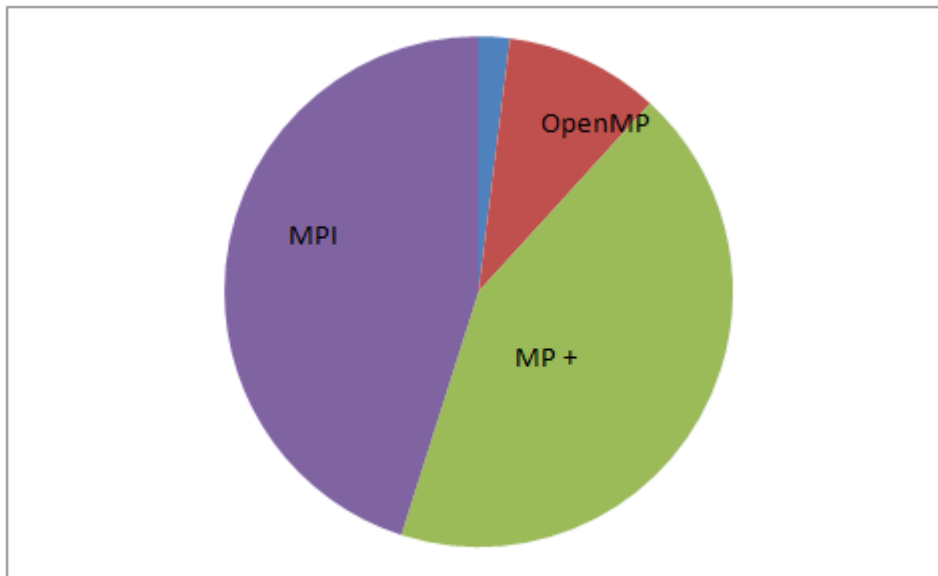


MATLAB PCT

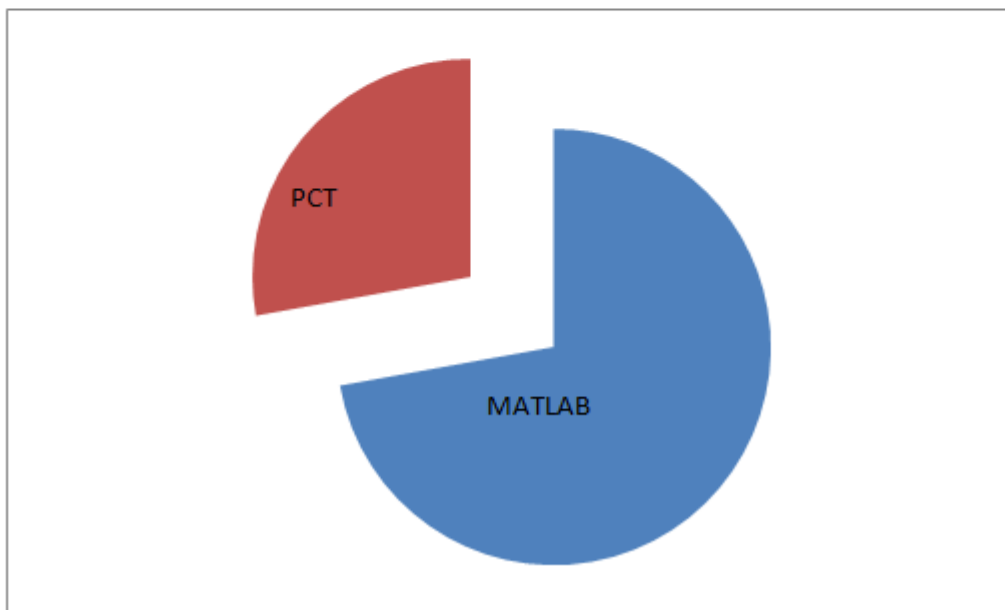
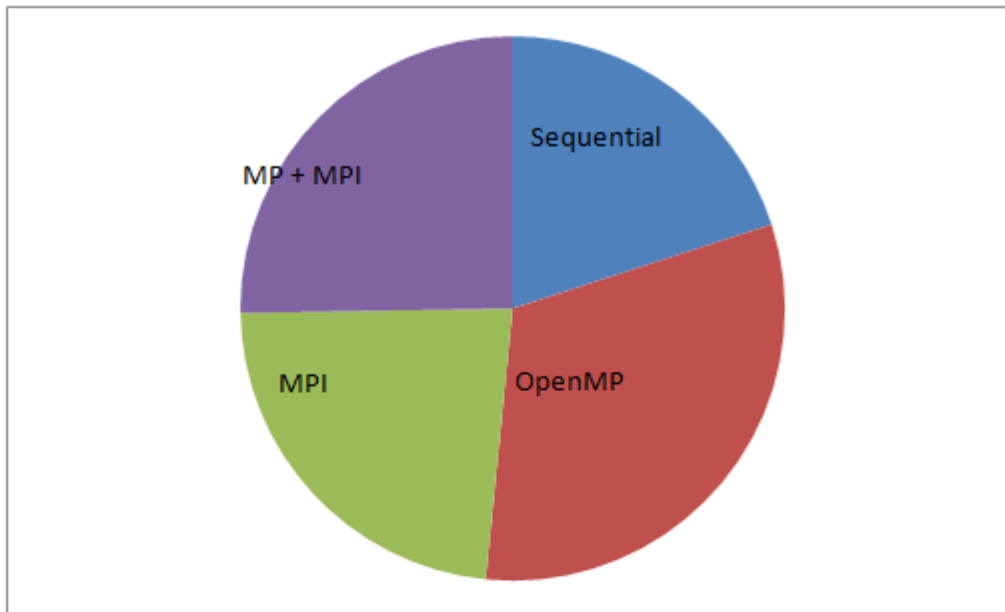


## Performance Comparison

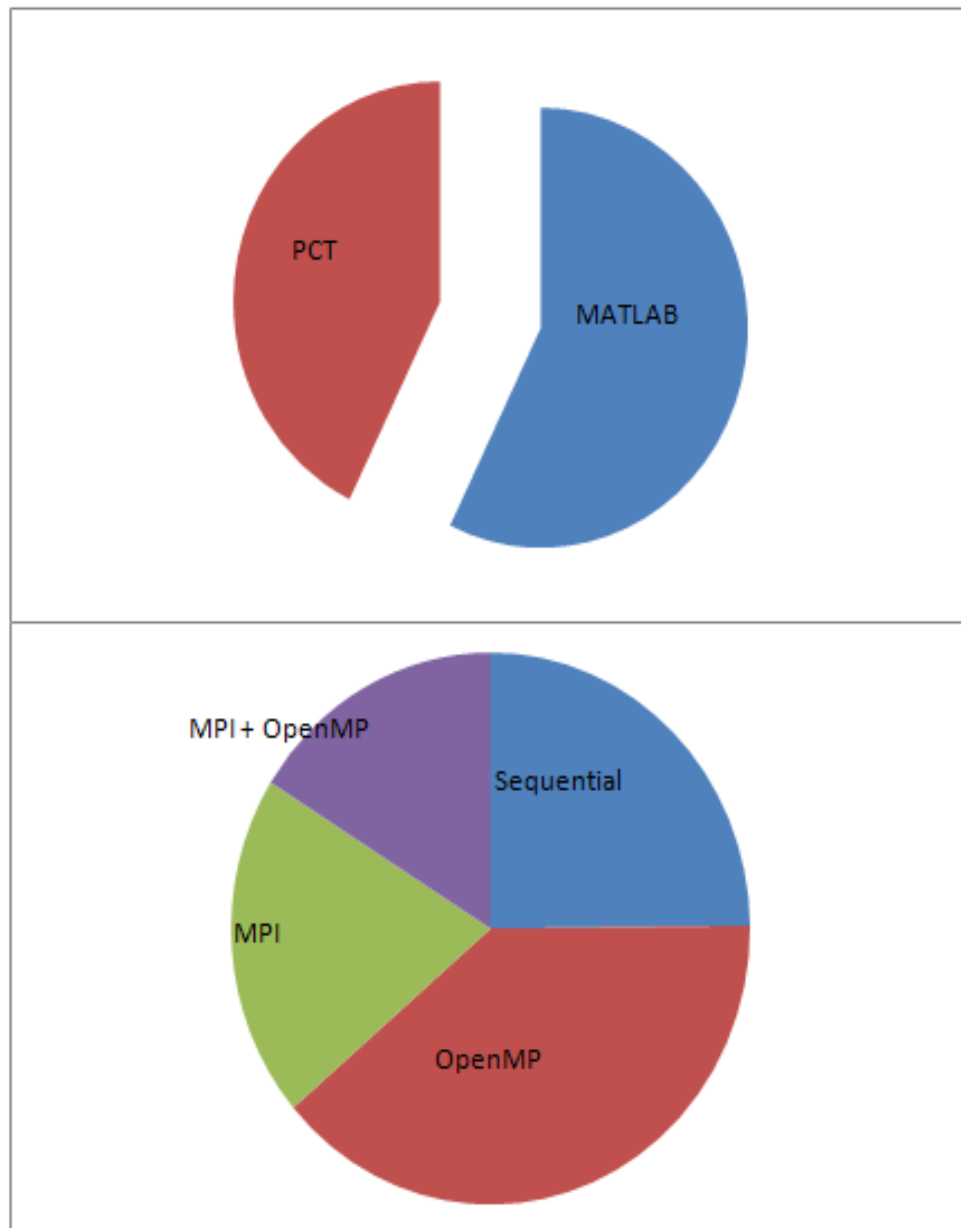
Alice.txt



world192.txt



gene.txt





## **5. ANALYSIS**

- The sequential code outperforms all forms of parallelization in execution time.
- This could be possibly of the fact as discussed earlier that string matching is not computationally intensive. So the parallelism intended ends up splitting the text into blocks and compares character which doesn't take anytime. Thus the overhead of launching threads and partitioning reduces that chances for speedup
- MPI does occasionally perform better than sequential code in cases of world192.txt and gene.txt. These may be due to inherent performance improvements present in the MPI\_File\_read\_at\_all function uses.
- The combination of OpenMP and MPI was expected to reduce execution time. However the results prove otherwise maybe due to the increased overhead.
- MATLAB both sequential and parallel implementation proved to be the worst in case of timing.
- MATLAB makes coding and debugging easier, more user friendly and easier to profile code. It however has tradeoff to ease in coding and that is reduce in performance.

## **7. DISTRIBUTION OF WORK:**

- Harishankar Patteri
  - MATLAB, C++ Sequential code
  - OpenMP
  - OpenMP+MPI version
- Vignesh Kumar Subramanian
  - C Sequential (alternate version)
  - MATLAB PCT
  - MPI

## **8. REFERENCE:**

- [1] Brute force string matching, Stoimen's web log  
<http://www.stoimen.com/blog/2012/03/27/computer-algorithms-brute-force-string-matching/>
- [2] Boyer-Moore-Horspool description [http://en.wikipedia.org/wiki/Boyer-Moore-Horspool\\_algorithm](http://en.wikipedia.org/wiki/Boyer-Moore-Horspool_algorithm)
- [3] Boyer-Moore-Horspool sequential code <https://github.com/FooBarWidget/boyer-moore-horspool>
- [4] Alternate code with traditional approach  
[http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore%E2%80%93Horspool\\_algorithm](http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore%E2%80%93Horspool_algorithm)
- [5] Overview of Discovery Cluster [http://nuweb12.neu.edu/rc/?page\\_id=27](http://nuweb12.neu.edu/rc/?page_id=27)