String Matching using MPI

CSE633 Spring 2019

PRESENTED BY NIRANJAN MIRASHI

UNDER THE GUIDANCE OF DR. RUSS MILLER AND DR. MATT JONES

CSE633

DATE – 04/25/19

Problem Definition

- Execute string matching in parallel on a very large string on a number of processors.
- Compare serial execution time and parallel execution time.

APPLICATIONS OF STRING MATCHING

- DNA SEQUENCING/BIO-INFORMATICS
- PLAGIARISM DETECTION
- SEARCH ENGINES
- INFORMATION RETRIEVAL
- DIGITAL FORENSICS AND MANY MORE..

SERIAL EXECUTION - KMP

- A linear time algorithm that solves the string matching problem by preprocessing P in $\Theta(m)$ time.
- Main idea is to skip some comparisons by using the previous comparison result.
- Uses an auxiliary array π that is defined as the following: $\pi[i]$ is the largest integer smaller than i such that $P_1 \dots P_{\pi[i]}$ is a suffix of $P_1 \dots P_i$

Pattern : A A B A

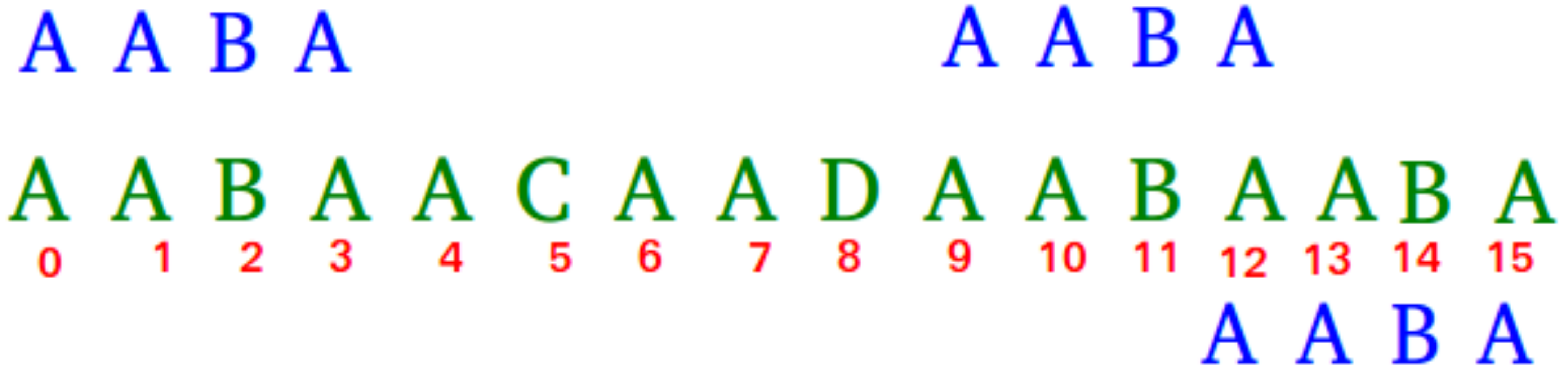


Fig: KMP String Matching

SERIAL EXECUTION TIME

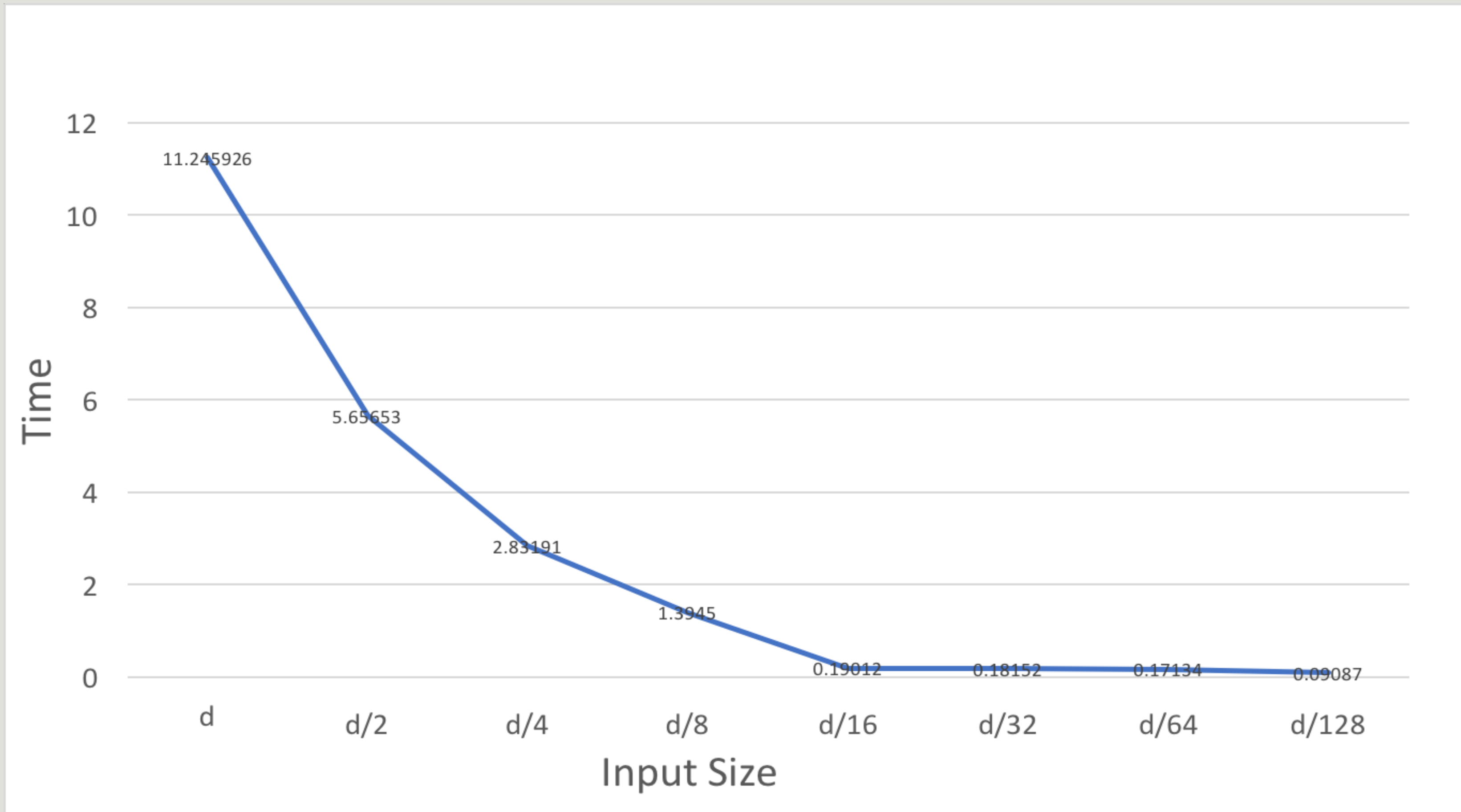
- **Test Parameters:**

String length (d) = 26 million

Pattern length = 9

INPUT SIZE	TIME (in seconds)
d	11.245926
d/2	5.65653
d/4	2.83191
d/8	1.3945
d/16	0.19012
d/32	0.18152
d/64	0.17134
d/128	0.09087

Serial Time Visualization



PARALLEL STRING MATCHING

- Divide and Conquer
- The main goal here is to divide the string equally among all the processors and to execute a string matching algorithm.
- We the execute a serial string matching algorithm on these sub-strings.

Algorithm

1. Distribute pattern to each processor

2. Divide string in a way that each processor has ‘d/n’ of string, where d = string.

3. Implement KMP serially. Keep a count of pattern occurrences.

4. If partial pattern in one processor, send length of partial pattern to next processor.

5. If processor receives a message containing length, check:
   - pattern[len(string)-len(partial_string)] at the start of its own respective sub-string.
   - If length matches, give current index - length(pattern) as output and increment count.

6. Once the above computation is done, compute prefix sum of counts of all processors. As a result, processor n will have the total count. Broadcast total count.

7. Output total count, execution time.

Sample Output

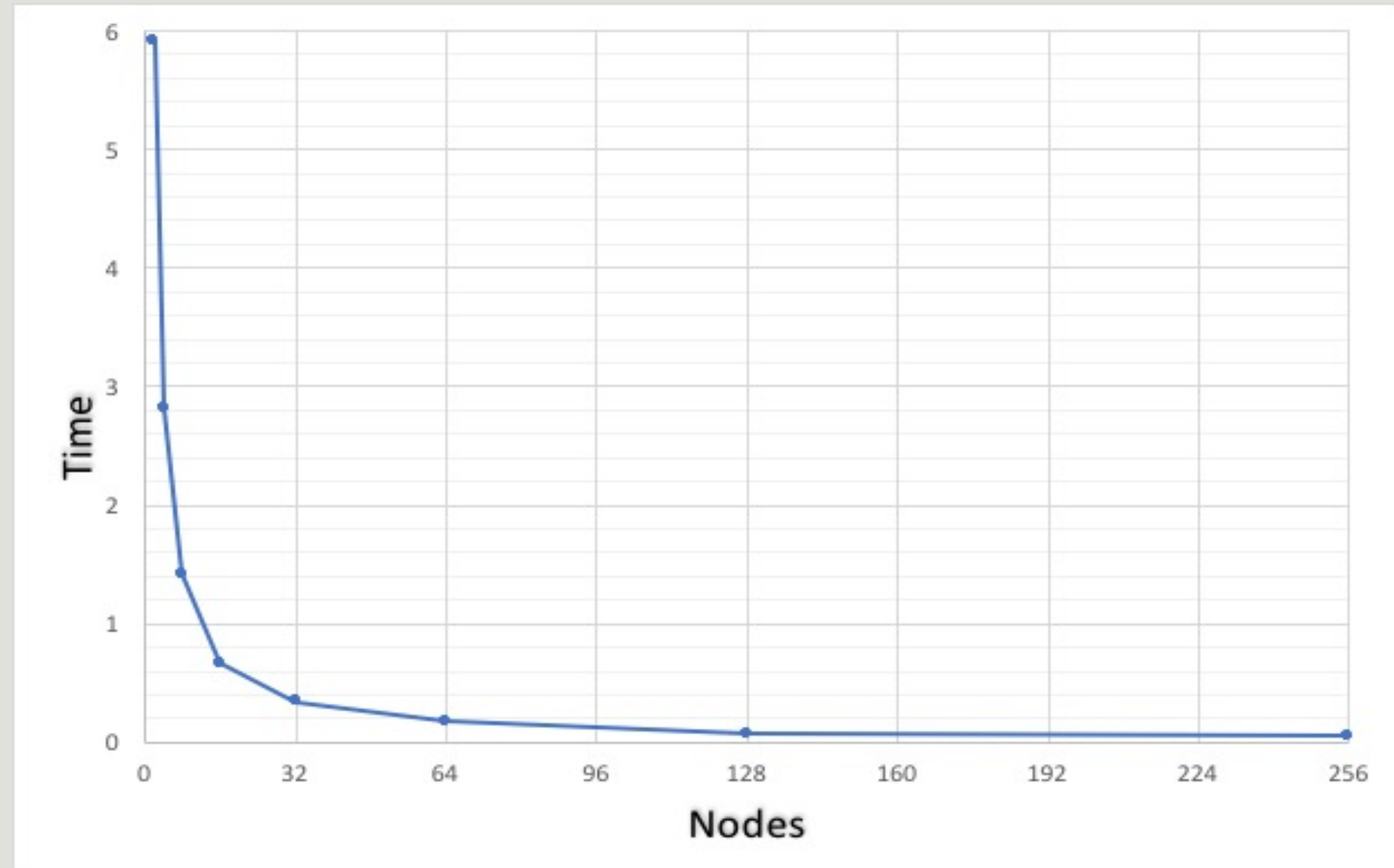
```
/util/common/python/py37/anaconda-2018.12/bin/python
Launch job
Processor 0 found pattern at index 116
Processor 0 found pattern at index 5325
Processor 0 found pattern at index 11654
Processor 0 found pattern at index 14943
Processor 0 found pattern at index 24873
Processor 0 found pattern at index 125492
Processor 0 found pattern at index 127637
Processor 0 found pattern at index 132446
Processor 0 found pattern at index 257597
Processor 0=  0.3231534957885742 seconds
Processor 21 found pattern at index 17301831
Processor 21=  0.32744717597961426 seconds
Processor 29=  0.314012506286621 seconds
Processor 8=  0.3192532062530517 6 seconds
Processor 31 found pattern at index 25597181
Processor 31=  0.322247028350830 1 seconds
Processor 24=  0.330214023590087 9 seconds
Processor 6=  0.3268842697143554 7 seconds
Processor 13=  0.31608128547668457 seconds
Processor 26 found pattern at index 21378136
Processor 26=  0.31433892250061035 seconds
Processor 7=  0.3238825798034668 seconds
Processor 3=  0.47931528091430664 seconds
Processor 14=  0.3257150650024414 seconds
Processor 10=  0.32370853424072266 seconds
Processor 16=  0.3344874382019043 seconds
Processor 5 found pattern at index 4731803
Processor 5=  0.32947230339050293 seconds
Processor 27=  0.32381367683410645 seconds
Processor 9=  0.3379390239715576 seconds
Processor 28=  0.31423163414001465 seconds
Processor 2=  0.3219468593597412 seconds
Total count = 17
Processor 4=  0.32674503326416016 seconds
Processor 1=  0.3231236934661865 seconds
Processor 12 found pattern at index 9789245
Processor 12=  0.3232133388519287 seconds
Processor 20=  0.3189244270324707 seconds
Processor 11=  0.3248271942138672 seconds
Processor 25=  0.32846593856811523 seconds
Processor 19=  0.31972289085388184 seconds
Processor 18=  0.3185272216796875 seconds
Processor 17 found pattern at index 14054590
Processor 17=  0.3219878673553467 seconds
Processor 30 found pattern at index 24034217
Processor 30 found pattern at index 24686640
Processor 30=  0.3194868564605713 seconds
Processor 23=  0.32378649711608887 seconds
Processor 22=  0.321062803268432 6 seconds
Processor 15=  0.31717896461486816 seconds
All Done!
"32.out" 75L, 3309C
```

Results

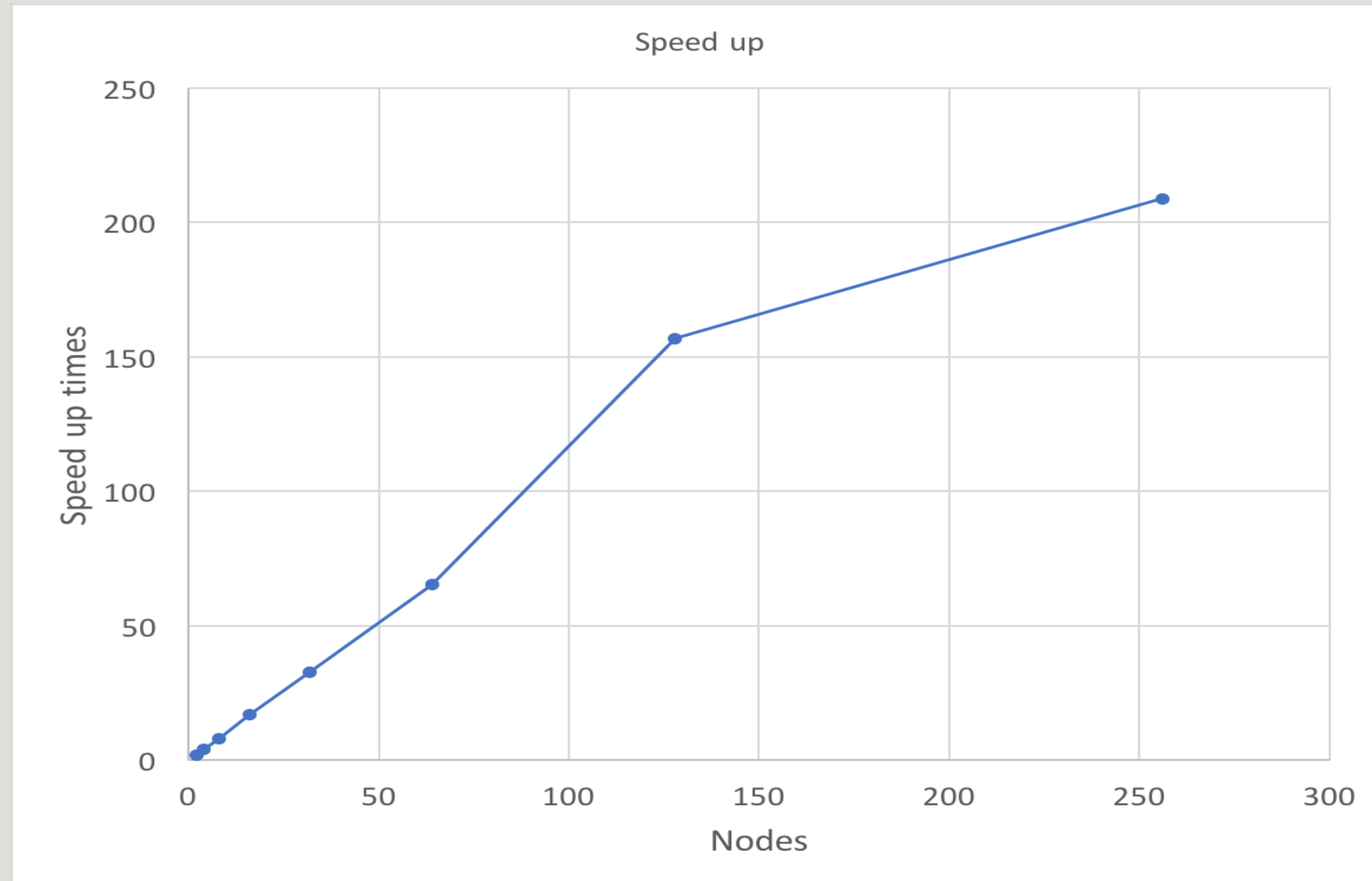
Test-case 1 :

- String of fixed length of 26 million chars.

Nodes	Time
2	5.912754
4	2.81644
8	1.420432
16	0.66441
32	0.34165
64	0.17203
128	0.07168
256	0.05385



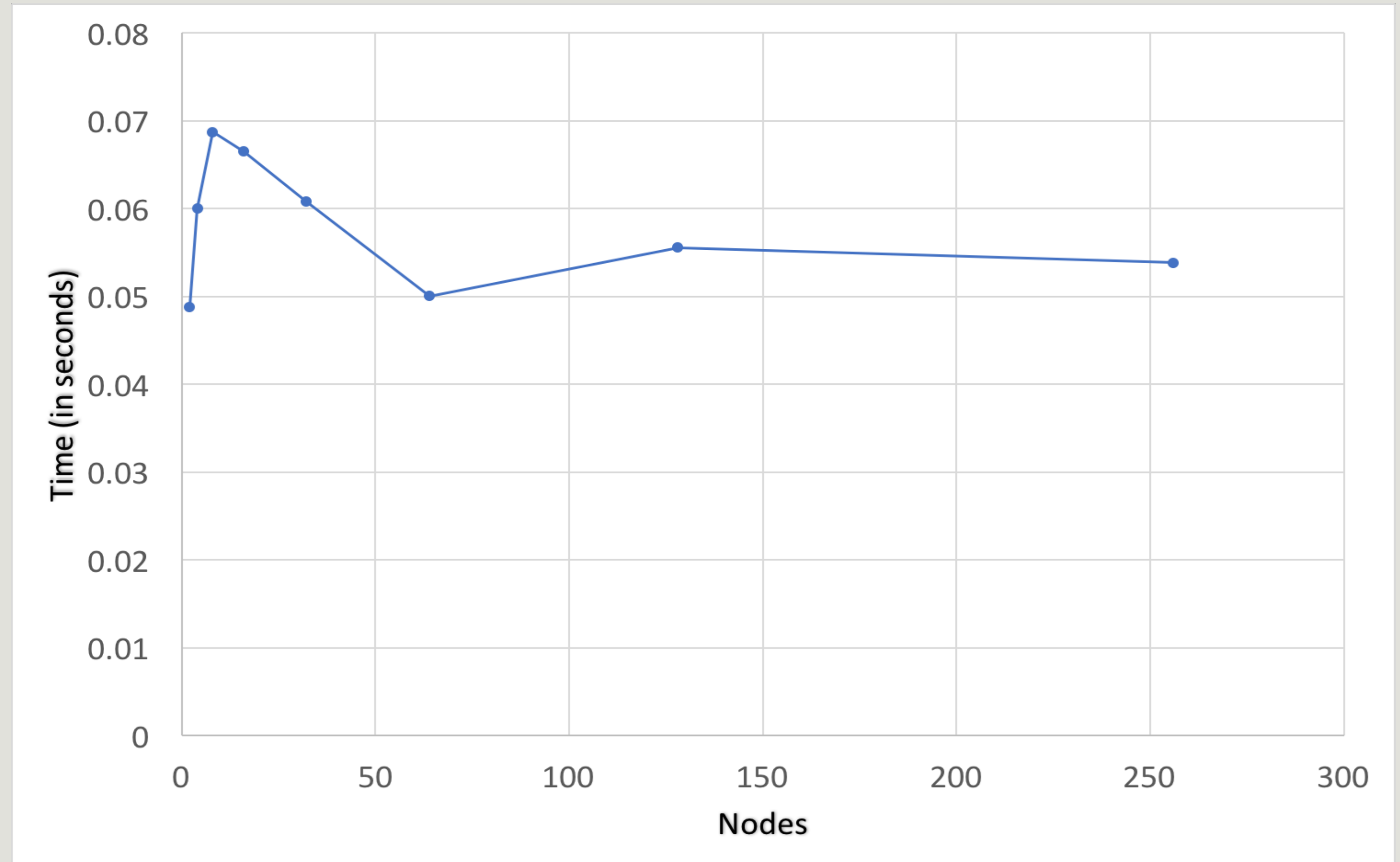
Speed-up



Test-case 2 :

- String of variable length.
- Size increases with number of processors.

Nodes	Time
2	0.048823
4	0.06
8	0.068705
16	0.066502
32	0.0608685
64	0.050034
128	0.05557
256	0.05385



Challenges

- The major task was to sequence processor communication efficiently.
- Another major task was to generate such a long string and edit it.
- Time taken to process data on 256 nodes was unpredictable.
- ‘mpi4py’ being recently developed, a deep insight on the package is not available.

Conclusion

- Time for executing a very long string of characters decreased exponentially with number of processors.
- Parallel implementation is recommended only for very large input strings.
- Time on reduced data on each processor showed linearity to an extent.
- MPI is very useful as it provides portability, efficiency and flexibility for running code.

Future work

- Executing string matching with very long pattern, approximately of (n-1) characters, where ’n’ is the length of String.
- Dividing pattern among processors.

THANK YOU.