Design and Performance Report

On

Parallel Document Frequency Index

by

SHIVANKIT GAIND	2015A7PS0076P
ROHIT GHIVDONDE	2015A7PS0077P

For the partial fulfilment of the
course on

Parallel Computing

Submitted to
Prof. Shan Sundar Balasubramaniam

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
April, 2018

Design

1. The problem of deriving document frequency is solved using **OpenMP tasks**.

2. A thread executing a task scans the directory assigned to it. While scanning it encounters two types of files:
    a. **Regular File** : Add the file path to the local vector of the thread
    b. **Directory File** : Create an OpenMP task to scan the directory. (This way the task is added to the task queue and is executed by some thread that gets free)

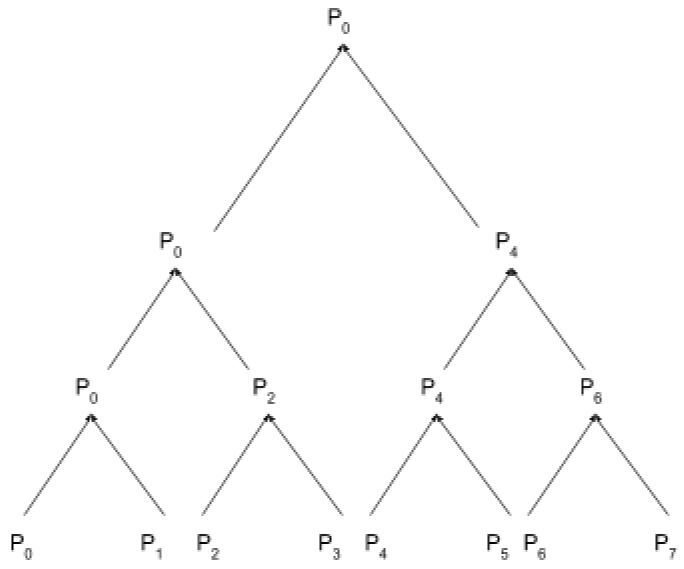
3. Thus each task involves creating more tasks. This way the file system tree is scanned and the files are added to the vectors(containing file paths).

4. The solution is divided into three phases:

    a. **Phase 1**: Each thread has a **local vector** containing the **paths to the files** encountered by the thread while implementing the tasks one by one. Each file is present in the vector of only one thread since each task is implemented only once (by a single thread).

    b. **Phase 2:** Then all the local vectors of threads containing file paths are merged into a single global vector parallely using **critical section** so that only one thread accesses global vector at a particular time (thus preventing race condition). These files in the global vector are then **dynamically** scheduled to the threads for processing in a parallel manner. Each thread has a **local unordered map** to store the document frequency of the words in the files processed by it. Thus each thread processes the files assigned to it and updates the document frequency of the words in its local unordered map.

    c. **Phase 3:** Once all the threads complete processing the files, the local maps of the threads are merged into a **global unordered map** using **parallel reduce**. Parallel Reduce helps in completing the map merging process in **O(log(N))** steps as compared to **O(N)** steps for sequential merging**.**



IDEA BEHIND USING LOCAL MAPS AND VECTORS:
Local maps and vectors are stored in the local stack memory of the thread which can be accessed faster as compared to the shared global map or vector stored in the process's stack memory. Also accessing shared global objects require concurrent access(for better performance) which can cause race conditions. Locking(for critical access) hampers performance. Thus local objects can be used which are accessed only by the corresponding threads thus removing the need for locking (mutual exclusion).

Cost of **parallel** execution = (Cost of **sequential** execution/**speedup**(number of threads)) **+**
(Cost for **spawning** threads + **dividing** work)

Performance Evaluation

The performance measurements for different values of p (where p is the number of cores used) for different datasets are given below:

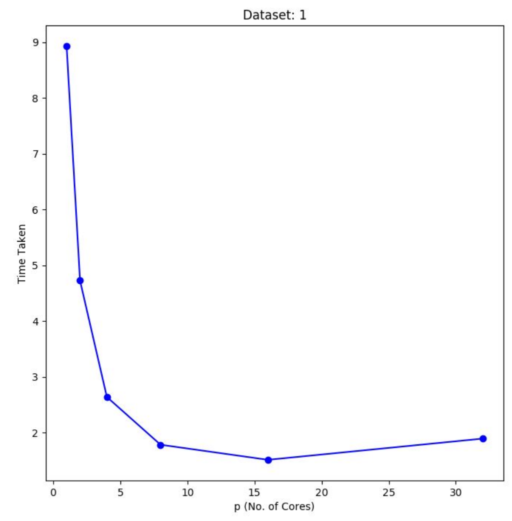
Datasets	**1**	**2**	**4**	**8**	**16**	**32**
1	8.931571	4.724588	2.63793	1.781824	1.511047	1.891073
2	0.60012	0.405933	0.322581	0.311905	0.268688	0.300755
3	2.961671	0.841133	0.487461	0.31939	0.300278	0.369074
4	12.912975	3.308967	1.900118	0.856267	1.023376	1.103591
5	286.696394	148.39645	68.101793	37.917743	24.12454	17.084709

The details for the datasets are given below:

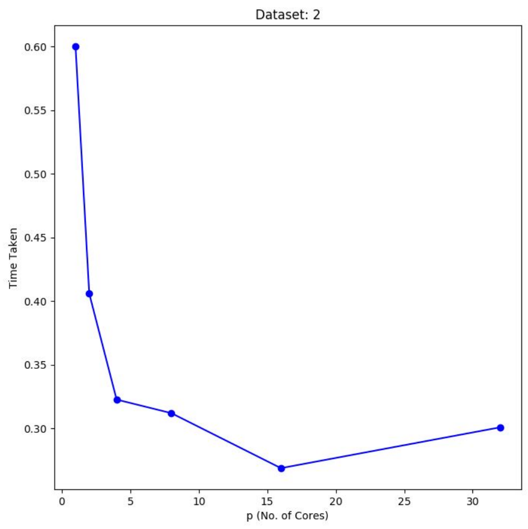
S.No	**Datasets**	**Max Depth**	**Avg Depth**	**Avg Branching Factor**	**Total Files**
1	20_newsgroups	2	2	973	20417
2	cacm	1	1	3204	3204
3	bbc-fulltext	3	2	318	2226
4	ohsumed-first-20000-docs	3	3	473	23166
5	simple	5	4	18	20165

The plots showing how performance improves with increase in the number of cores for each dataset are given below:

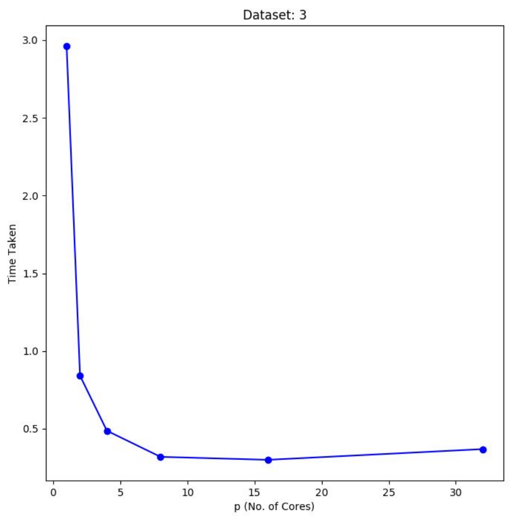
It is clear that performance improves with utilization of more cores. However, for datasets 1-4, the time slightly increases towards the end as we increase the number of threads. This is because the **overhead of thread creation and dividing work among them** is more than the **work done per thread**(threads spend a lot of time doing I/O). This behaviour is not observed in case of Dataset 5 since it’s a huge dataset and has files of **large sizes**, hence, the tasks are **cpu intensive** and work done per thread is high.



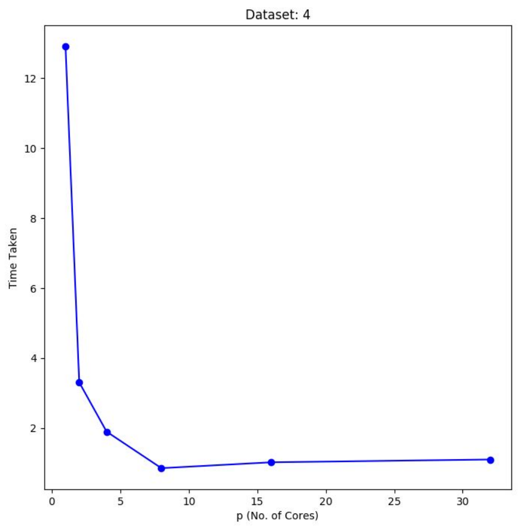
Dataset: 1
Time Taken
p (No. of Cores)



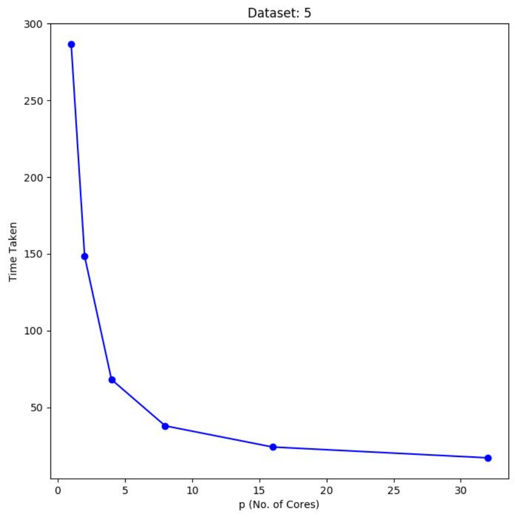
Dataset: 2
Time Taken
p (No. of Cores)



Dataset: 3
Time Taken
p (No. of Cores)



Dataset: 4
Time Taken
p (No. of Cores)



Also, a super linear speedup is observed in some cases when we increase the number of threads from 1 to 2 since the tasks are i/o intensive as well.

The following are some of the measurements taken for different depths and different number of cores used. The measurements are taken for a branching factor of 2.

Total Number of Files = 100

Depth/p	1	2	4	8	16	32
5	0.172852	0.106177	0.0542	0.028484	0.018798	0.015292
6	0.196482	0.106003	0.057204	0.0286	0.017435	0.014737
7	0.13706	0.072734	0.040384	0.022311	0.01355	0.01498

Total Number of Files = 1000

Depth/p	1	2	4	8	16	32
5	1.199678	0.635494	0.368826	0.22506	0.13483	0.078697
6	1.140142	0.663775	0.373614	0.226967	0.139198	0.078553
7	1.141046	0.683848	0.338524	0.248286	0.136823	0.066642

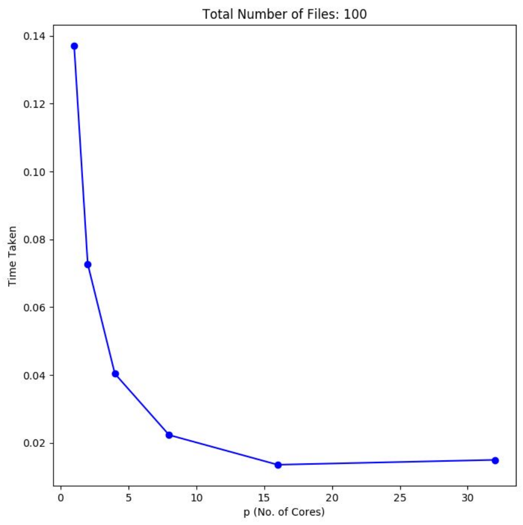
Total Number of Files = 10000

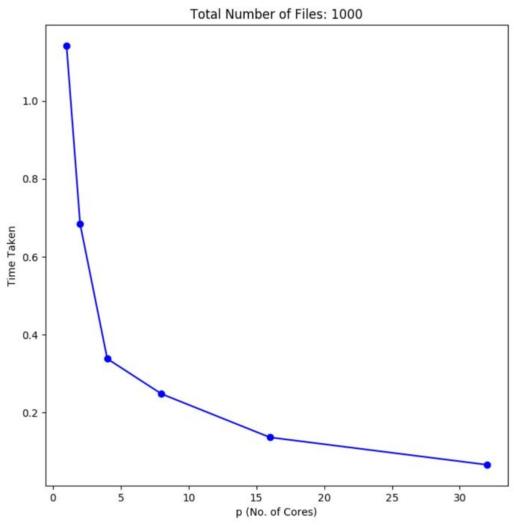
Depth/p	1	2	4	8	16	32
5	11.028924	5.782345	2.767128	1.495616	0.808244	0.46628
6	11.119174	5.723495	2.854544	1.463369	0.781435	0.44007
7	11.00879	5.710087	2.856994	1.463497	0.834132	0.488429

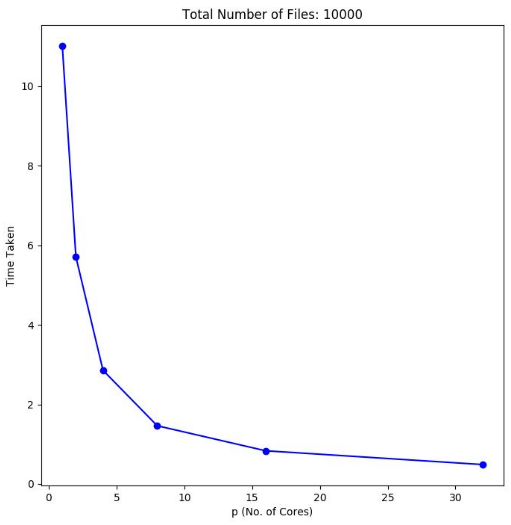
Total Number of Files = 100000

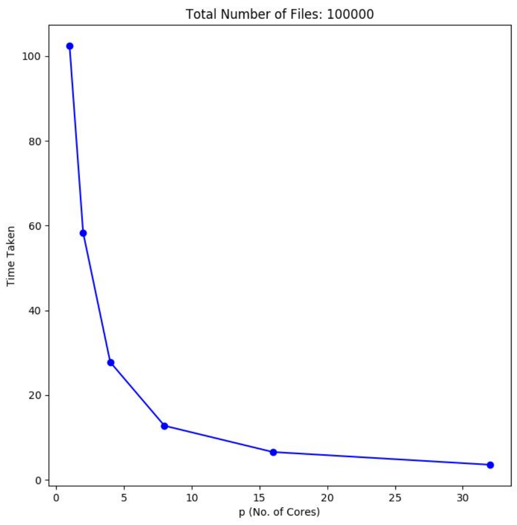
Depth/p	1	2	4	8	16	32
5	110.823677	56.807278	26.686036	13.179107	6.678949	3.540475
6	110.18018	56.823435	26.262763	12.865237	6.586503	3.548085
7	102.471816	58.375649	27.773314	12.75713	6.555118	3.552686

The following plots show how performance improves with more utilization of cores at depth of 7 and branching factor of 2:





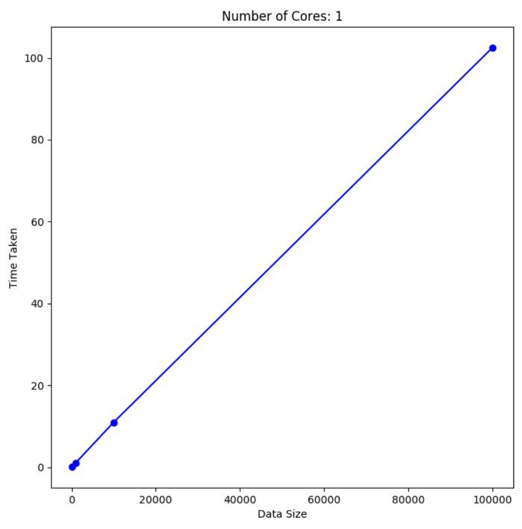


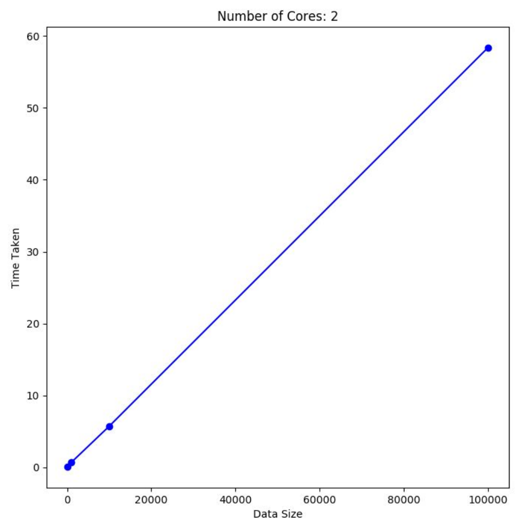


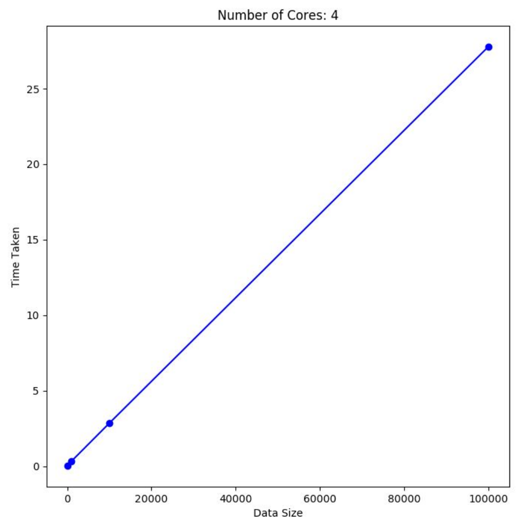
It is again clear that performance improves with utilization of more cores from 1 to 32.

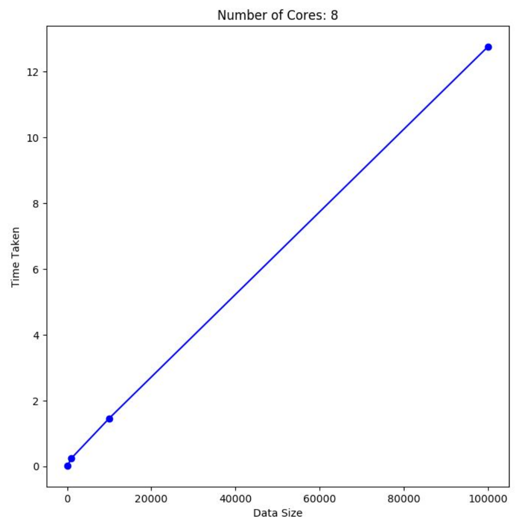
The following table and plots show how time increases as the data size i.e the number of files increase keeping the number of cores constant (with average depth = 7 and average branching factor = 2):

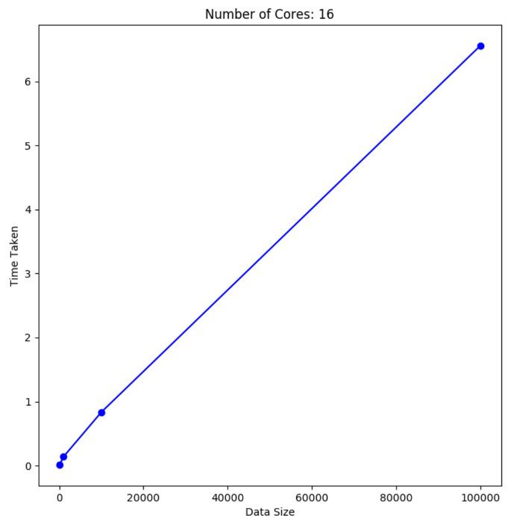
Total Files/ p	**1**	**2**	**4**	**8**	**16**	**32**
100	0.13706	0.072734	0.040384	0.022311	0.01355	0.01498
1000	1.141046	0.683848	0.338524	0.248286	0.136823	0.066642
10000	11.00879	5.710087	2.856994	1.463497	0.834132	0.488429
100000	102.471816	58.375649	27.773314	12.75713	6.555118	3.552686

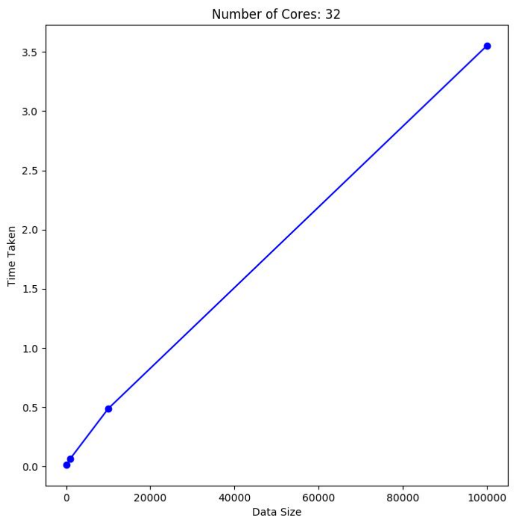

Number of Cores: 1
Time Taken
Data Size
0
20
40
60
80
100
0
20000
40000
60000
80000
100000


Number of Cores: 2
Time Taken
Data Size
0
10
20
30
40
50
60
0
20000
40000
60000
80000
100000


Number of Cores: 4
Time Taken
Data Size
0
5
10
15
20
25
0
20000
40000
60000
80000
100000


Number of Cores: 8
Time Taken
Data Size
0
2
4
6
8
10
12
0
20000
40000
60000
80000
100000





Also, keeping the total number of files and cores utilized constant, the time doesn't change much with change in average depth of the hierarchy. The following table is built for number of files equal to 1000 and 100000 with branching factor fixed at 2 and number of threads fixed at 4. The depth is varied from 5 to 7.

Depth/Total Files	**1000**	**100000**
5	0.368826	26.686036
6	0.373614	26.262763
7	0.338524	27.773314