COL380 Assignment 2 Part-2: PageRank using MapReduce

Pratyush Pandey (2017EE10938), Harkirat Dhanoa (2017CS50408)

4^{th} May, 2020

Contents

1 Introduction

PageRank Algorithm comprises of repeated mulltiplications of a vector to a matrix until the vector converges (becomes *stationary vector*). Usually matrix mulltiplication takes running time $O(npq)$ if A is a n x p matrix, B : p x q and hence C : n x q. Therefor multiplying an n x n matrix to an n x 1 vector would naively take $O(n^2)$.
We now make use of the fact that the n x n matrix is a sparse one. Instead of computing the product of every number if each row, we only iterate over the non-zero values. This makes the running time $O(m)$ where m is the number of hyperlinks (edges in the graph) across all n web pages.

2 Parallel Map Reduce Characteristics

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits or shards. The input shards can be processed in parallel on different machines.
Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specifed by the user.
The illustration below shows the overall fow of a MapReduce operation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in the illustration correspond to the numbers in the list below).

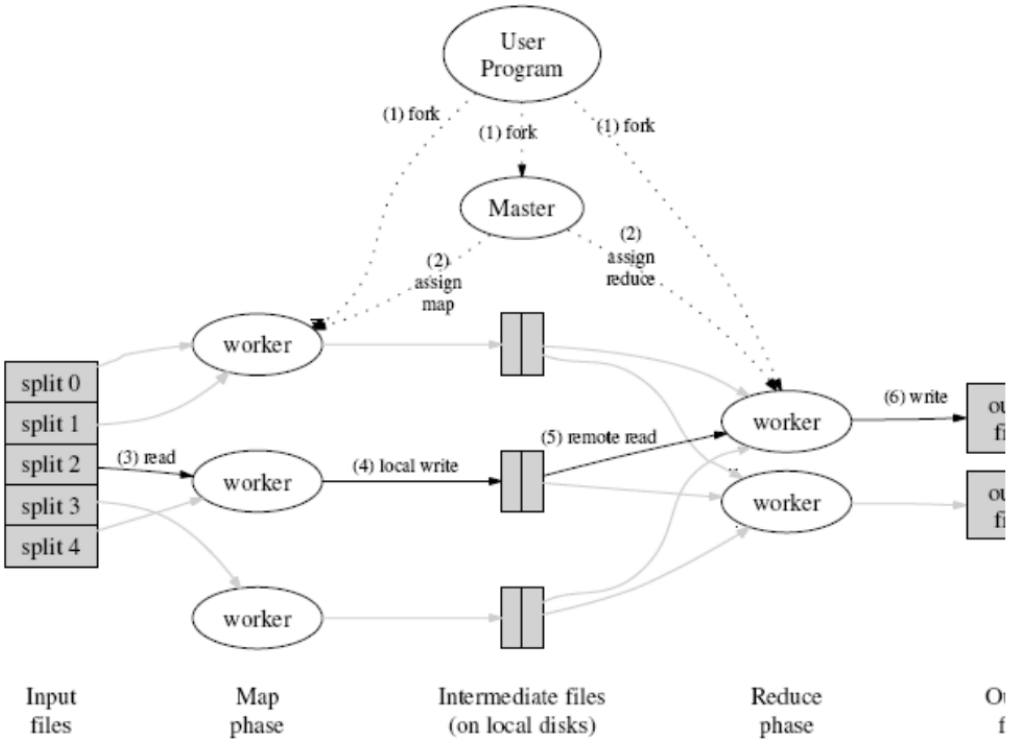


Figure 1: MapReduce in Parallel - Strategy

- A worker who is assigned a map task reads the contents of the corresponding input shard. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
- The locations of these buffered pairs are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
- When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. If the amount of intermediate data is too large to fit in memory, an external sort is used.
- The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
- When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code

3 MapReduce Strategy used for computing PageRank

3.1 Map function

Takes as input a pair of key (int) and value ($vector < int >$) which is basically a row indexed by the key.

Returns pair(s) of key (int) and value (double) which indicate that the page represented by the key is getting an update in its importance indicated by the value
Basically, for every b in value: emit(b , page_rank[key]/num_hyperlinks[key])

3.2 Reduce function

Takes as input a pair of key (int) and value ($list < double >$) which is basically list of updates to the importance of page indexed by key from all pages which refer to it
Returns pair(s) of key (int) and value (double) which indicate that the page represented by the key is getting a combined update in its importance indicated by this value which is sum of all numbers in the input list called value
Basically, for every b_i in value: emit(key, $\sum_{i=1}^{n} b_i$)

4 RunTime Analysis

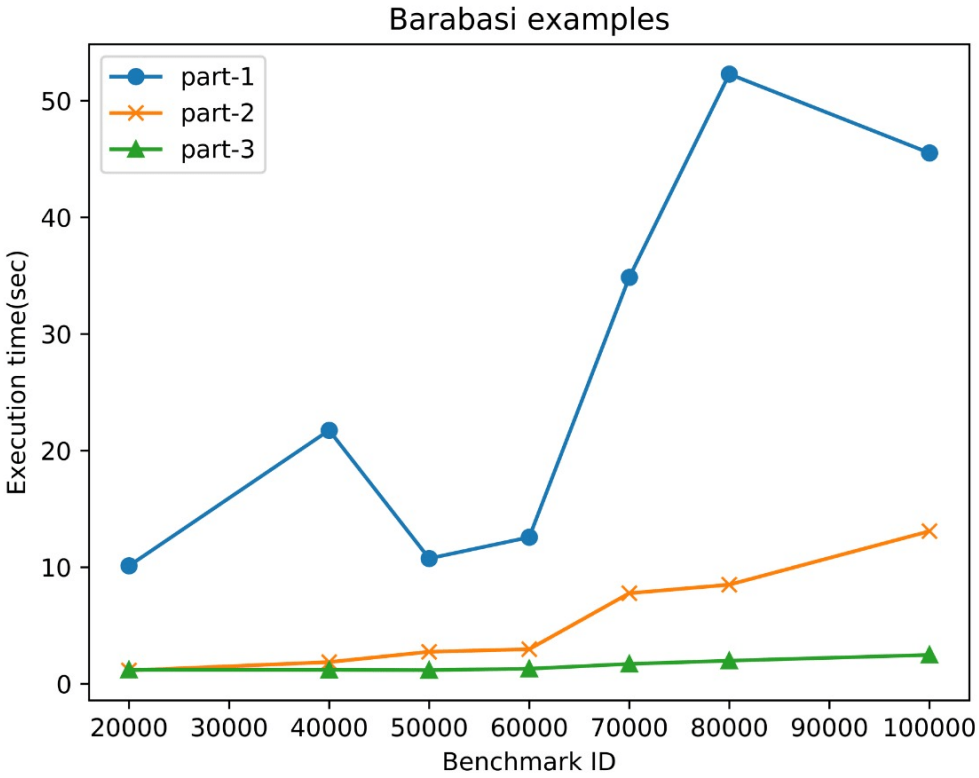


Figure 2: Runtimes analysis for Barabasi Benchmark

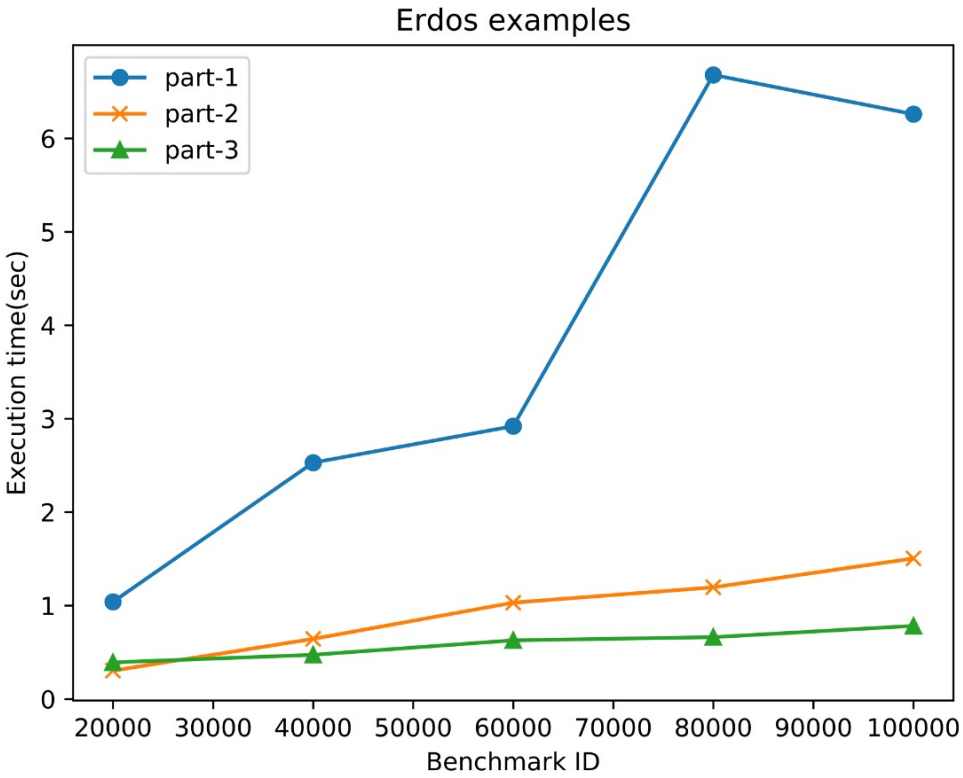
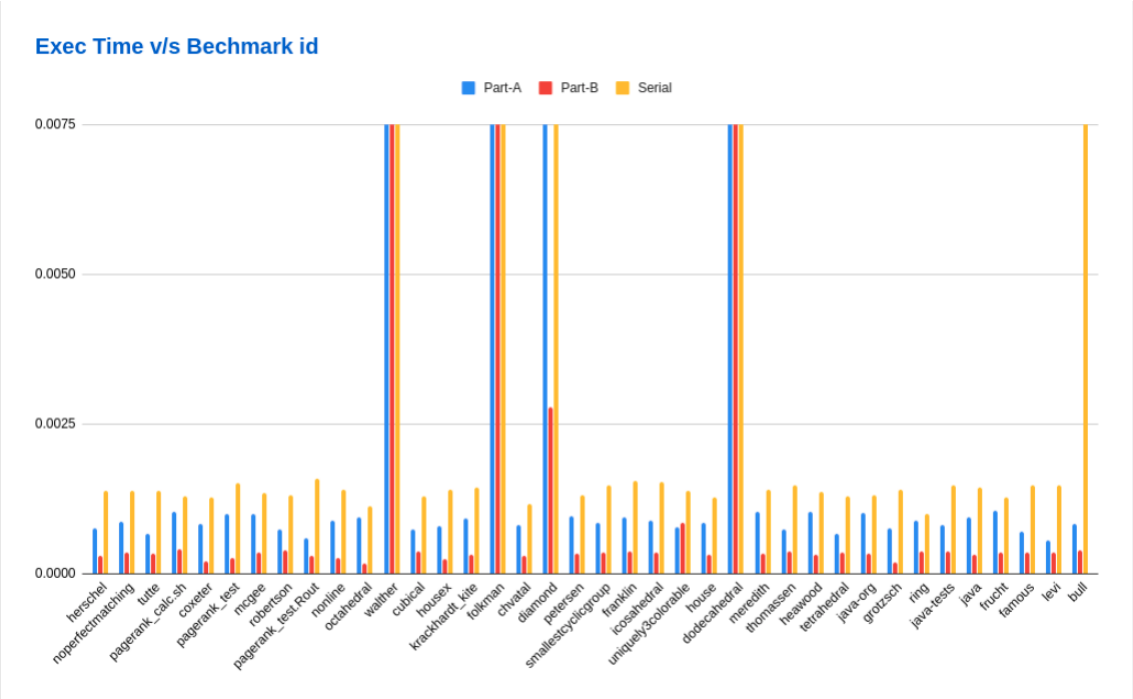
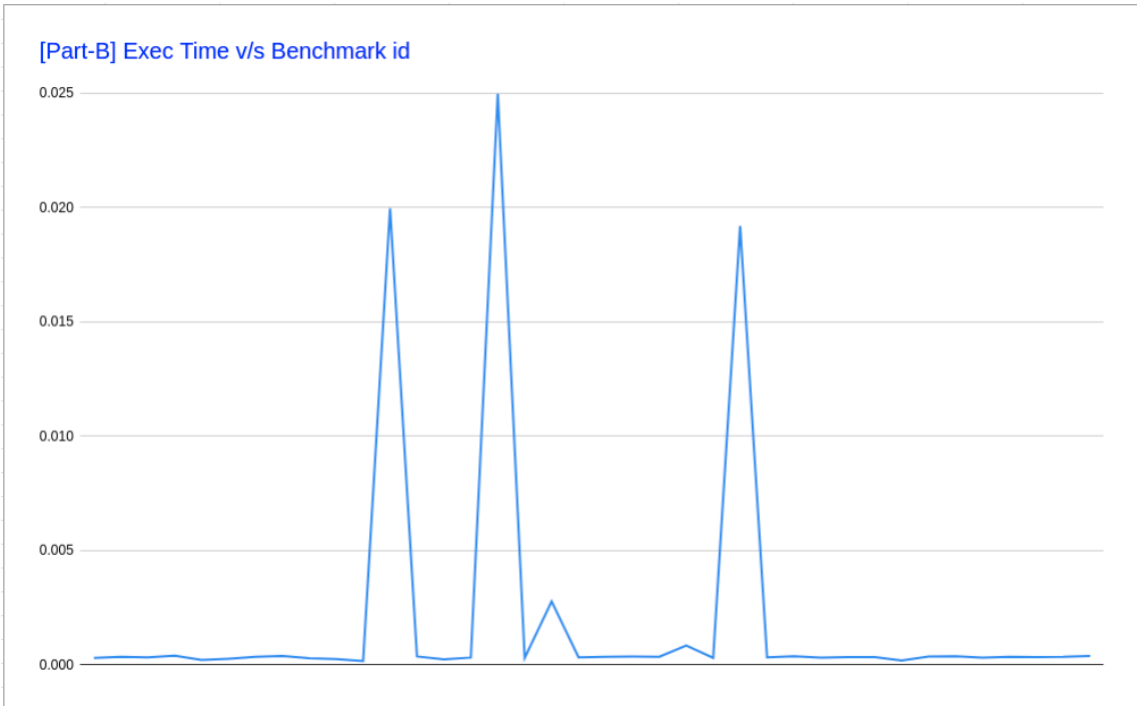
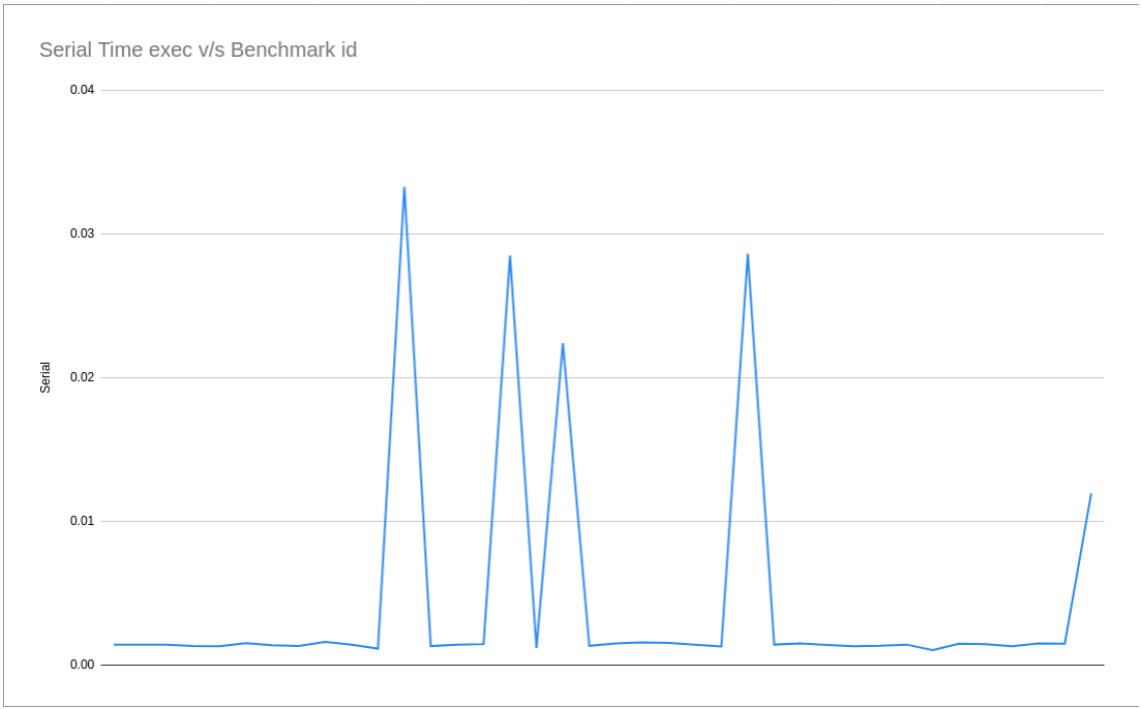


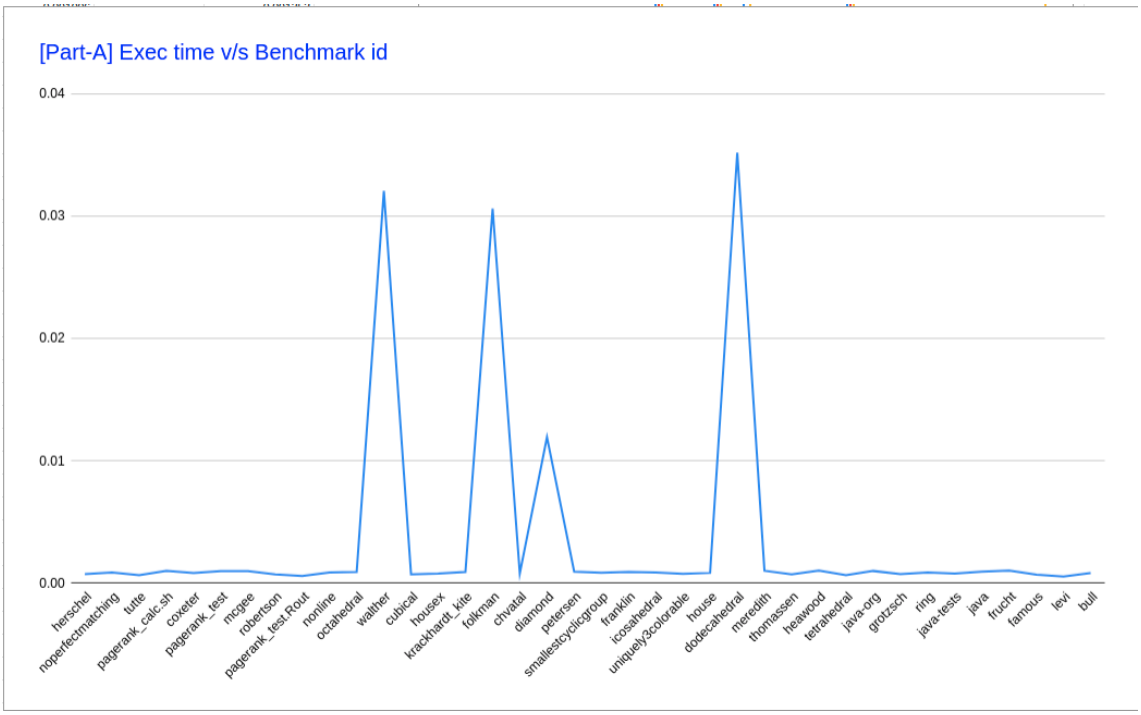
Figure 3: Runtimes analysis for Erdos Benchmark

Benchmark id	Part-B	Part-A	Serial
herschel	0.00031	0.000753	0.001393
noperfectmatching	0.000355	0.00088	0.001395
tutte	0.000333	0.00067	0.001387
pagerank_calc.sh	0.000409	0.001033	0.001303
coxeter	0.000218	0.000844	0.001278
pagerank_test	0.000273	0.001002	0.001507
mcgee	0.000354	0.001006	0.001353
robertson	0.000392	0.000741	0.001306
pagerank_test.Rout	0.000302	0.000603	0.001589
nonline	0.000258	0.000897	0.001399
octahedral	0.000176	0.000942	0.001121
walther	0.019971	0.032093	0.033288
cubical	0.000376	0.000742	0.001296
housex	0.000245	0.000802	0.001396
krackhardt_kite	0.000326	0.000925	0.001437
folkman	0.024986	0.030653	0.028489
chvatal	0.000306	0.000809	0.001168
diamond	0.002782	0.011949	0.022399
petersen	0.000339	0.000962	0.001316
smallestcyclicgroup	0.000361	0.000857	0.001475
franklin	0.000369	0.00094	0.001559
icosahedral	0.000359	0.000884	0.001528
uniquely3colorable	0.000851	0.000783	0.001388
house	0.000313	0.000856	0.001267
dodecahedral	0.019196	0.035227	0.028618
meredith	0.000338	0.001035	0.001405
thomassen	0.00038	0.000734	0.001483
heawood	0.000326	0.00104	0.001371
tetrahedral	0.000349	0.000671	0.001291
java-org	0.000347	0.001012	0.001315
grotzsch	0.000198	0.000764	0.001401
ring	0.000369	0.00089	0.001009
java-tests	0.000379	0.000809	0.00147
java	0.000322	0.000946	0.001437
frucht	0.000356	0.001056	0.001285
famous	0.000351	0.000712	0.001486
levi	0.000358	0.000561	0.001473
bull	0.000395	0.000842	0.011936


Exec Time v/s Bechmark id
Part-A
Part-B
Serial
0.0075
0.0050
0.0025
0.0000
herschel
noperfectmatching
tutte
pagerank_calc.sh
coxeter
pagerank_test
mcgee
robertson
pagerank_test.Rout
nonline
octahedral
walther
cubical
housex
krackhardt_kite
folkman
chvatal
diamond
petersen
smallestcyclicgroup
franklin
icosahedral
uniquely3colorable
house
dodecahedral
meredith
thomassen
heawood
tetrahedral
java-org
grotzsch
ring
java-tests
java
frucht
famous
levi
bull


[Part-B] Exec Time v/s Benchmark id
0.025
0.020
0.015
0.010
0.005
0.000





5 Observations

5.1 mapreduce C++ library

Here we have used the C++ library for mapreduce using a single machine. The running time for this code was slower due to the fact that it uses 1 process exclusively for data gathering and collection two times (one for mapper and one for reducer call) while all the remaining processes actually implement the map and reduce on the provided data.

5.2 Own mapreduce library

Here we see significant rise in performance due to the fact that we have specified an even distribution of work across all the running processes. Although the intermediate data sorting has been done on the basis of node number this does not necessarily guarantee equal workload for all the processes.

5.3 mapreduce MPI library

In this method we see further improvement of performance than our own MPI mapreduce library because of the fact that here we are also distributing evenly the sizes of value-list for each key i.e. instead of just giving equal fraction of total number of keys to each process, we are in fact also taking into account the time it'll take to process those key. Here the assumption that each key takes equal time to be reduced is not made.

6 Optimisations

6.1 Performance

The distribution of data and computing division across multiple processors offers many advantages:

- With MPI it is required less effort in terms of the timing required for data handling, since each process has its own portion.
- MPI offers flexibility for data exchange.

6.2 Compiler Optimisations

Used compiler flags:
-flto -march=native -mtune=knl -O3 -Ofast
What they do?

- flto - Link Time Optimization (LTO) gives GCC the capability of dumping its internal representation (GIMPLE) to disk, so that all the different compilation units that make up a single executable can be optimized as a single module. This expands the scope of inter-procedural optimizations to encompass the whole program (or, rather, everything that is visible at link time).
- march=native - adds all optimization flags applicable to the hardware I'm compiling on. To check which flags are actually being activated do gcc -march=native -Q –help=target ...
- O3 - (use -Ofast instead) It enables all -O3 optimizations. It also enables optimizations that are not valid for all standard compliant programs.

NOTE: O3 flag isn't used in compile times due to artificial speedups and loss of parabolic $O(n^2)$ runtime graphs.

Appendix

The following is the machine information on which the execution run times were calculated
model name : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Ram : 16GB
Number of cores : 4