CMPE 478
Parallel Processing

A Parallel Version of Google Ranking Process Using OpenMP Documentation

FALL’2022 - Project 1 Report

Artun Akdoğan - 2020400327

Hüseyin Seyyid Kaplan - 2015400033

1. Details of the Algorithm/Implementation

○ Purpose Of This Program:

This program ranks graphs using a parallel google ranking algorithm. This program compares different scheduling algorithms with different parameters, and returns top 5 results from the algorithm with sufficient logging on the console.

○ Explanation Of This Program:

This program is optimized with parallel programming by OpenMP. First, it reads the file into a string pair vector. Also, it identifies unique elements by an unsorted_map. Then, it identifies each unique string with an index. After that, it pairs string pairs with a 2-dimensional vector. Please note that 2-dimensional vectors are not fully allocated; thus, it is space efficient. This procedure is done in parallel, but not reported as not requested. Still, it drops the runtime of that part from 20 seconds to 8 seconds.

Those values are then used to initialize the CSR vector. The values vector is the non-zero elements in the matrix. col_indices vector is the indices to the values vector in a row. row_begin vector points to the beginning of the row at the col_indices vector. While the values vector doesn't have to be sorted, col_indices and row_begin vectors should be in sorted order for this program to work. Also CSR_Matrix class holds arr_dict vector and two_vec_diff variables to set after running the “ops” function. Those values are public, and not essential for the class to work. They only exist for outer usage, especially in the main.cpp file.

CSR_Matrix class can be initialized by the parser function at parser.h file or it can be initialized from a file structured in csv format. Also, this class has a native function to write its contents to a file for later loading from it. ops function is a matrix multiplication function of a matrix and a vector, optionally multiplying the result with the desired value, and summing it to another value if required. This function returns relevantly sized result vectors.

main.cpp file contains the code to iterate over different scheduling options with different chunk sizes and different thread numbers and keep a log of each iteration. Also, it initializes the CSR matrix by calling the class, looping to run the "ops" function until the vector difference is less than the specified (1e-6) epsilon, and then getting and reporting the top 5 elements of the result.

○ Extra Arguments For The Program:

i. ./program load backup.csv

Reads CSR matrix from specified file. (You can change backup.csv file)

ii. ./program save backup.csv

Initialises CSR matrix from local graph.txt file, then writes it to specified file. (You can change backup.csv file)

iii. ./program

Initialises CSR matrix from local graph.txt file. No other file is interfered with.

- Files Read and Written On Runtime:
  i. **graph.txt**

     This is the web graph file. It should be in the same directory with the program. It is read and parsed to create a CSR matrix. Please refer to Erdos Web Graph.

  ii. **log.csv**

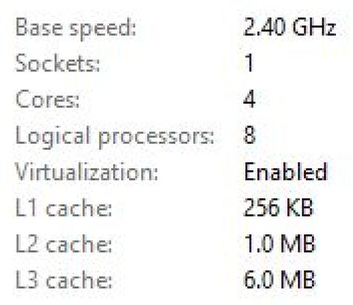
     CSV file that is written to show runtime speed comparison.

  iii. **result.csv**

     Top 5 results from running the program.

2. Details of the Machine/CPUs

- Machine Specifications:
- Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz



Base speed:	2.40 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB

- 16 GB DDR3 RAM
- WSL2 Ubuntu 22.04 on Windows 10

- Application Compile Instructions:
- To run this application, you need to install “libomp-dev” library:
  - sudo apt install libomp-dev
- Compile the program using g++ with openmp library:
  - g++ -std=c++14 -fopenmp -DNDEBUG main.cpp -oprogram
- Give execution permission to the file:
  - chmod +x ./program
- Ensure that graph.txt is in the same directory with the compiled program.
- Then, run the application with desired arguments:
  - ./program

3. Timings Table (Generated CSV Table)

- Timing Table For Parallel Unoptimised Code:

Test No.	Schedulin	Chunk Siz	No of Iter	1	2	3	4	5	6	7	8
1	static	1	14	8.585946	5.472939	4.069867	3.480593	3.06505	2.701412	2.541459	2.250747
2	static	100	14	8.793549	4.882185	3.535615	3.011892	2.705045	2.323199	2.106022	2.166908
3	static	10000	14	8.703169	5.438621	3.905824	3.892272	3.672617	3.532637	3.44504	4.003998
4	static	1000000	14	8.850818	8.342254	7.589463	7.761777	7.421728	7.658976	7.406114	7.619735
5	dynamic	1	14	8.884985	6.950629	5.837165	5.510514	4.719574	4.313229	3.978245	3.969155
6	dynamic	100	14	9.377205	4.702197	3.301103	2.590726	2.328131	2.203948	2.059179	2.002204
7	dynamic	10000	14	8.605611	5.457623	4.018326	3.493604	3.878156	3.089577	2.983481	2.942167
8	dynamic	1000000	14	8.32657	7.142899	7.163261	7.710156	7.549937	7.609684	7.583592	7.549552
9	guided	1	14	8.799982	4.707702	3.441802	2.665951	2.410827	2.144611	2.161038	1.927636
10	guided	100	14	8.731619	4.686011	3.400028	2.790485	2.427519	2.313286	2.09816	2.245059
11	guided	10000	14	8.560977	5.277798	3.810693	3.318927	2.769767	3.198299	2.649902	2.822533
12	guided	1000000	14	8.628206	7.539397	7.659844	7.862643	7.476336	7.453376	7.475303	7.751837
13	auto	1	14	9.203473	7.599511	7.239165	7.089604	6.762502	6.615006	6.620957	6.201433
14	auto	100	14	8.7354	7.555003	7.374851	6.93526	6.983525	6.925881	6.909685	6.34712
15	auto	10000	14	8.56542	7.536989	7.017665	6.863595	6.854117	6.386527	6.397801	6.336157
16	auto	1000000	14	8.128904	7.043293	6.767327	6.59318	6.393437	6.249321	6.0678	6.132602

- Timing Table For Parallel Optimised Code (O1):

Test No.	Schedulin	Chunk Siz	No of Iter	1	2	3	4	5	6	7	8
1	static	1	14	2.963262	2.283651	1.754398	1.658058	1.487491	1.399127	1.050637	1.089171
2	static	100	14	3.024112	1.667171	1.313315	1.200081	1.025262	0.955049	0.881241	0.932425
3	static	10000	14	3.02369	2.027762	1.501255	1.39255	1.313154	1.299876	1.175295	1.206103
4	static	1000000	14	3.045481	2.8265	2.847193	2.802126	2.87965	2.819608	2.911352	2.928187
5	dynamic	1	14	3.430854	2.704985	2.126251	1.88333	1.67088	1.605396	1.637962	1.499841
6	dynamic	100	14	3.042942	1.664496	1.21081	1.03091	0.925655	0.969767	0.840929	0.800116
7	dynamic	10000	14	3.223215	2.051046	1.531534	1.260224	1.207583	1.175455	1.161815	1.128043
8	dynamic	1000000	14	2.915439	2.713025	2.760607	2.731478	2.980692	2.841504	2.803911	2.917396
9	guided	1	14	3.035148	1.705502	1.232356	1.013881	0.866761	0.807715	0.939245	1.053828
10	guided	100	14	3.04214	1.698179	1.362754	1.047127	1.054837	0.959051	0.826737	0.802872
11	guided	10000	14	3.147623	1.926228	1.444184	1.22611	1.256022	1.198821	1.15684	1.146935
12	guided	1000000	14	3.092323	2.917184	2.869997	2.722409	2.758121	2.743642	2.975752	2.859909
13	auto	1	14	2.986709	2.72469	2.860528	2.878527	2.619876	2.592182	2.43089	2.418152
14	auto	100	14	3.036436	2.943864	2.755664	2.779919	2.639821	2.546384	2.446706	2.481643
15	auto	10000	14	3.016443	2.827817	2.672531	2.713438	2.678798	2.534931	2.503991	2.357798
16	auto	1000000	14	3.067884	2.74527	2.779789	2.655223	2.690547	2.573058	2.535199	2.391219

4. Discussion of Results

- Discussion of the results:

No.	Nodes	Scores
1	0491md82hej8u15vi98isrmuih	0.005989
2	3165mii1s1g0invqs94q303v0v	0.005193
3	4mekp13kca78a3hfsrb0k813n9	0.004252
4	2494c7mt12frm3c3go86abe13h	0.002917
5	1o4ivpkrnqiisvdivocv1n7jav	0.001576

Scores were close to each other for the first 5 elements. However, results were consistent between each iteration of different schedule algorithms and block sizes. So, it can be said that parallelisation was successfully applied.

- ## Discussion Timings Table:

First of all, as it can be seen from the timing table for unoptimised code, as the thread number increases, nearly always run time has decreased. For unoptimised code, it was far more evident between run time results.

Secondly, it seems that as chunk size decreases, run time decreases except for auto schedule. Nevertheless, on dynamic and static schedules, for very small chunk size, it seems that run time is higher because of the overhead, especially on dynamic.

Finally, in general, a dynamic schedule seems to be better on the right chunk sizes. For static, overheads seem to be lower, and for guided, it shows its best performance on chunk size 1.