CS-342: Parallel Programming

Homework 1 *2023*
Report

Author: *Minos Stavrakakis* - csd4120

Table of Contents

1. Execution - Running Instructions

Compilation / Execution

- Use Makefile located inside repo:

  `make all` compiles code and creates ***cs342_ass1.exec*** executable

  `make clean` removes the executable

- To run the program type:

  `./cs342_ass1.exec <filename>`

Execution flags

Programm makes use of various execution flags:

`-threads` followed by # of threads (*1 - 4, valid values*) explicitly sets the number of threads that will be created. *4 being the default value.*

`-datastats` will create a ***data_stats.txt*** file, depicting the data topology in a viewer friendly manner.

`-timestats` will create (and append in subsequent runs) a ***time_stats.txt*** file, recording the time 'spent' in the parallel section of the code.

Example:

`./cs342_ass1.exec Datasets/Facebook/facebook_combined.txt -threads 4 -datastats -timestats`

The above command uses every flag mentioned above

2. Datafiles

Program demonstrates the page rank algorithm in a parallel environment and in order to do so, makes use of a set pool of input data. The datafiles that were used are:

Email-Enron.txt UNDIRECTED GRAPH, # nodes: 36692

facebook_combined.txt UNDIRECTED GRAPH, # nodes: 4039

p2p-Gnutella24.txt DIRECTED GRAPH, # nodes: 26518

and can be found at http://snap.stanford.edu/data/index.html .

Data example from Email-Enron.txt file

```
# FromNodeId    ToNodeId
.....

7351    22493
7352    195
7353    195
7353    492
7353    934
7353    1145
7353    1899
7353    1909
7353    4839
7353    7429
7354    116
7354    195
7354    662
7354    1371
7354    7355
7355    116
7355    195
7355    662
7355    1371
7355    7354
7356    155
7356    195
.....
```

Due to the fact that each datafile presents the information in a unique style, the program is tailor-made to the aforementioned datafiles and will **NOT** work for any other input aside those three.

3. Parsing / Data Structs

Parsing of datafiles happens in a serial manner. Each line gets tokenized and appends information in a pre-allocated pointer array. The array, named `node_arr` inside the code, has 1 - 1 corresponding size to the number of nodes of the specific datafile being used and points to a user defined `node` struct presented below.

```
typedef struct node_s {
    int64_t id; // = -1; @ init
    double_t rank; // = 1.0; @ init
    double_t rank_to_give; // = -1.0; @ init
    cvector_vector_type(int64_t) vec_nbor_in; // = NULL; @ init
    uint16_t v_in_sz; // = 0; @ init
    cvector_vector_type(int64_t) vec_nbor_out; // = NULL; @ init
    uint16_t v_out_sz; // = 0; @ init
} node;
```

**cvector_vector_type( ) is a macro from the open source cvector.h library, found in https://github.com/eteran/c-vector, which implements the basic functionalities of a C++ Vector type in C language.*

4. Algorithm - Implementation

The Algorithm used to calculate the ranking statistics of each node is a simplified version of the page-rank algorithm used by Google to calculate site traffic in the early days of internet (https://en.wikipedia.org/wiki/PageRank).

The formula used to calculate the rank of a node **X** in an iteration ***i*** is:

$$\mathbf{PR}_i(X) = 0.15 + 0.85 \times \sum_{n=1}^{\substack{\#\text{ of incoming} \\ \text{edges of X}}} \frac{\mathbf{PR}_{i-1}(Y_n)}{\substack{\#\text{ of outgoing} \\ \text{edges of Y}}}$$

*where **Y**, if it exits, is a neighboring node with outgoing edge to **X**.*

Translating the above formula into C threaded code we have:

```
void *
page_rank_thrd(void *myargs) {

    thread_args *t_arg = (thread_args*) myargs;
    // Transfer args to local vars for eou
    int64_t t_BGN = t_arg->BGN;  // BGN & END signify the node_arr partition
    int64_t t_END = t_arg->END;  // that each thread will manipulate

    double_t sum_from_in_nbors;
    uint16_t vi;
```

```

    /* Calculate rank_to_give for Iteration #1 */
    for (int64_t j = t_BGN; j <= t_END; ++j) {
            // node_arr[j].rank_old = node_arr[j].rank_new;
            node_arr[j].rank_to_give = node_arr[j].rank / ((double_t) node_arr[j].v_out_sz);
        }

    /* BARRIER */
    // make sure rank_to_give have initialized
    pthread_barrier_wait(&thread_barrier);

    /* WHILE BEGIN */      // NO_ITERATIONS 500
    for (size_t it=0; it < NO_ITERATIONS; ++it) {

        for (int64_t i = t_BGN; i <= t_END; ++i) {

            sum_from_in_nbors=0.0;

            for (vi=0; vi < node_arr[i].v_in_sz; ++vi) {

                sum_from_in_nbors += node_arr[node_arr[i].vec_nbor_in[vi]].rank_to_give;
            }

            /* Calculate Rank */
            node_arr[i].rank = BASE_RANK + ( DAMPING_FACTOR * sum_from_in_nbors);
            /* Calculate Rank to disperse to nbors */
            node_arr[i].rank_to_give = node_arr[i].rank / ((double_t) node_arr[i].v_out_sz);
        }

        /* BARRIER */
        // make sure all new PR's finished calculating
        pthread_barrier_wait(&thread_barrier);

    /* WHILE END */
    }

    return NULL;
}
```

The above implementation gives a time complexity of $O(n^2)$ and a spatial complexity of O(n) for every iteration.

5. Statistics

All times are measured in seconds				
file: *Email-Enron.txt*				
# threads	1	2	3	4
	1.600478	1.145046	1.080822	0.849641
	1.580826	1.23923	0.985046	0.889048
	1.736304	1.287158	1.142245	1.068432
	1.760355	1.333816	1.003084	1.00685
	1.67555	1.2556	1.0993	0.948224
	1.666273	1.343019	0.948089	0.986691
	1.704076	1.259243	1.086598	0.918895
	1.610218	1.369597	1.142961	0.98255
	1.646344	1.207235	1.105968	1.003201
	1.599059	1.19579	1.110067	1.042179
average	1.6579483	1.2635734	1.070418	0.9695711
avg deviation	0.0505633	0.05585928	0.055007	0.05449528
std deviation	0.0617027899766291	0.0711298820659159	0.0676989136627103	0.0682467568467543
avg speedup	1	1.3121107962545	1.54887931630447	1.70998114527135
file: *facebook_combined.txt*				
# threads	1	2	3	4
	0.301964	0.17909	0.17002	0.130102
	0.285096	0.207231	0.168975	0.13708
	0.301243	0.181159	0.169923	0.169116
	0.311608	0.176436	0.216092	0.147544
	0.345294	0.19496	0.203274	0.141937
	0.328148	0.21751	0.199359	0.133342
	0.298482	0.152512	0.172498	0.158918
	0.338895	0.212481	0.17215	0.138719
	0.316276	0.195477	0.217224	0.176082
	0.29133	0.201387	0.205657	0.128217
average	0.3118336	0.1918243	0.1895172	0.1461057
avg deviation	0.01625572	0.01562004	0.018804	0.01344744
std deviation	0.0201687191803986	0.0197107095427277	0.0205421483935401	0.0166223307033373
avg speedup	1	1.62562094583429	1.64541054848848	2.13430139960316
file: p2p-Gnutella24.txt				
# threads	1	2	3	4
	0.351575	0.235093	0.197444	0.171177
	0.379116	0.273781	0.189523	0.219208

All times are measured in seconds				
	0.317877	0.240808	0.22297	0.201744
	0.514587	0.263344	0.209191	0.145275
	0.54929	0.273555	0.189095	0.179123
	0.502376	0.279163	0.243247	0.205991
	0.413205	0.232365	0.219116	0.167514
	0.449497	0.306088	0.263684	0.162452
	0.505519	0.232071	0.235942	0.19925
	0.437547	0.289873	0.24143	0.162089
average	0.4420589	0.2626141	0.2211642	0.1813823
avg deviation	0.0565408181818182	0.0200217090909091	0.0184458181818182	0.0183025090909091
std deviation	0.0766686160381881	0.0263156804858498	0.0250947759937756	0.0238190688130796
avg speedup	1	1.68330222939286	1.99878144835376	2.43716669156803

- Since file parsing and data structure initialization happens in a non-threaded, serial way, the statistics shown in the table above regard only the parallel section of the program: *i.e.* START 'stopwatch' just before thread declaration and initialization and STOP measuring after `pthread_join()` has returned.
- Wall-clock time was measured using the POSIX function `clock_gettime()`, defined in time.h.
- The formula used to calculate the avg speedup is:

$$avg\ speedup = \frac{avg\ 1\ thread\ execution}{avg\ n\ threads\ execution}$$

6. Epilogue

..I just wanted to mess with Typora... :P