Presentation Document

Authors: Enrico Alberti s279434, Enrico Castelli s280124, Benedetto Giulivi s269827
Date: 2021-02-14

Contents

- Presentation Document
- Contents
- Abstract
- Code schema
- Main data structures
  - `graph => row_g[vertex_num]`
  - `labels => row_l[vertex_num].field[labels_num]`
  - `queries => queries[queries_num].vertex[source, dest]`
- Time and memory usage
- How to run

Abstract

Our group developed the Q2 project which consists of implementing an algorithm named **GRAIL** used for Scalable Reachability Index for Large Graphs. The project could be divided in three main steps:

1. **Reading** the Directed Acyclic graph (**DAG**)
2. Generating the required **labels**
3. Testing the **query reachability** using the labels

Code schema

DAG read

The main thread (**main.c**) prepares the thread's data structure for the **DAG** read. In particular, it reads the first line of the input file DAG which contains the vertex number size used to allocate the right amount of memory. The DAG file is read (**readGraph.c**) by the maximum number of threads that the processor is able to support without scheduling. For instance if a processor is quad-core and it's able to support 8 threads, eight is the number of threads that we create. The idea is to split the file into `NUM_THREADS` equal parts and let each thread read (without protection because there are no problems in reading) and fill its own part of the *shared* DAG structure. Moreover, just after the building of the DAG structure, with some *synchronization*, each thread counts how many **roots** there are in its 'local' fragment and then all together (with some *protection*) fill a *shared* array containing all roots. These roots will be used later for label generation.

DAG read - What and How

- MAX threads allowed by processors without scheduling
  - Each thread takes care of the number of vertexes between *inf* and *sup*.
  - Since the file is not homogeneous, we cannot divide it for the vertexes number. So each thread takes care of `FILE_SIZE/NUM_THREADS` bytes of the whole graph.
  - Each thread estimates the **inf** as `(FILE_SIZE*ThreadID)/NUM_THREADS` and **sup** as `(FILE_SIZE*(ThreadID+1))/NUM_THREADS`
  - But in this way the thread doesn't know in which offset of the line it is. So the thread has to read until it finds '\n' to estimate its 'inf' and 'sup' correctly.

Roots initialization - What and How

- A first barrier is used to wait for the end of the DAG read.
- A second barrier is used to wait for the count of the roots.
- A third barrier is used to wait for the main thread to allocate the memory for the roots array.
- A mutex is used to protect the insertion of the roots in the *shared* array.

Label generation

In the next step, the main thread re-initializes the threads data structure for the **labels** generation (**buildLabels.c**). In general, we follow the paper structure. Here, we run in parallel *one thread for each label*. Each of these threads *visits* in random order the *roots*, and recursively visits -in random order- their *children* as described by the GRAIL algorithm.

We tried to implement a parallel basic version in which each thread also generates one *thread for each child* and with the required protection, they explore the graph to generate the label. We removed this implementation due to the limit of the maximum number of threads we are able to create (`PTHREAD_THREADS_MAX`) which is easily reached in a very large graph.

Another attempt we did was to generate `NUM_THREADS-NUM_LABELS` threads that visit the DAG in parallel, in addition to one thread per label. Since the memory usage is greater because of the threads data structure and the time is greater as well since a protection for `rank_root` is required, we decided to not use this version.
We also tried to split the label generation among `num_roots/NUM_THREADS` threads, but this solution did not provide any improvement in terms of memory and time as well.

Label generation - What and How

- 1 thread for each label : current version
  - each thread runs its own label generation - many labels at the same time
- MAX threads allowed by processors without scheduling for each label
  - divide the roots number by many threads - only 1 label at each time
  - mutex for each node to protect many threads working on the same node
  - mutex for shared rank_root
- 1 thread for each child : limit of threads exceeded (20,000)
  - mutex for each node to protect multiple threads working on the same node
  - mutex for shared rank_root
- 1 thread for each label + remaining threads without scheduling for roots visit : no improvement
- mutex for each node to protect multiple threads working on the same node
  - mutex for shared rank_root

Query resolution

Finally, in the last step the main thread re-initializes the threads data structure for the **query** reachability test. In particular, the main thread counts the number of queries to allocate the right amount of memory. Then the query resolution begins (**solveQuery.c**). The idea is to divide the queries in `NUM_THREADS` equal parts (so that all the cores work without scheduling) and let each thread solve its 'local' subset applying the logic described in the GRAIL paper algorithm (without the use of the exception list). At the end a log file is created to report the results.
As a reminder, given a query V1->V2, we have to check if the labels of V2 are NOT contained in the labels of V1, if so we can conclude that V1 cannot reach V2. Instead, if the labels of V2 are contained in the labels of V1 ([a, b] ⊆ [c, d]) we cannot claim that V1 can reach V2 but we have to do a DFS with pruning such that for each child C of V1 that doesn't contain V2's labels, we don't follow that path. Otherwise, we follow that path and verify the reachability.

Query resolution - What and How

- MAX threads allowed by processors without scheduling
  - Query file read by the main thread alone because of its limited size.
  - Queries divided in equal parts (`NUM_QUERIES/NUM_THREADS`) within the number of threads
    - no protection because each thread writes in its local part (`array_queries[j].can_reach`)

- The reachability of the queries is written by the main thread alone because of its limited size.

During each of these main phases, the main thread evaluates the **time** and **memory** usage needed to execute the code.

Main data structures

`graph => row_g[vertex_num]`

```c
typedef struct row_graph {
  int edge_num;
  bool not_root;
  int *edges;
} row_g;
```

So we will have something like:

- vertex 0 has children x, y
- vertex 1 has children h, k

Structure	Description
`row_g[0]`	vertex number 0
`row_g[1]`	vertex number 1
`row_g[0].edges[0]`	children 'x' of vertex 0
`row_g[0].edges[1]`	children 'y' of vertex 0

`labels => row_l[vertex_num].field[labels_num]`

```c
typedef struct row_label {
  int* lbl_start;
  int* lbl_end;
  bool* visited;
} row_l;
```

So we will have something like:

- vertex 0 has 2 labels -> [x, y], [h, k]
- vertex 1 has 2 labels -> [v, w], [z, a]

Structure	Description	Interval
`row_l[0]`	labelS of vertex number 0	
`row_l[0].lbl_start[0]`	begin label 0 of vertex 0	[x,
`row_l[0].lbl_end[0]`	end label 0 of vertex 0	y]
`row_l[0].lbl_start[1]`	begin label 1 of vertex 0	[h,
`row_l[0].lbl_end[1]`	end label 1 of vertex 0	k]
`row_l[1].lbl_start[0]`	begin label 0 of vertex 1	[v,
`row_l[1].lbl_end[0]`	end label 0 of vertex 1	w]
`row_l[1].lbl_start[1]`	begin label 1 of vertex 1	[z,
`row_l[1].lbl_end[1]`	end label 1 of vertex 1	a]

`queries => queries[queries_num].vertex[source, dest]`

```
typedef struct el_list_query {
  int num[2];
  bool can_reach;
} el_query;
```

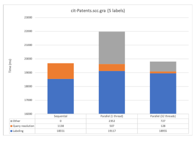
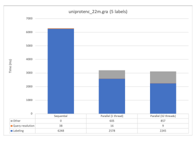
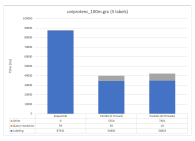
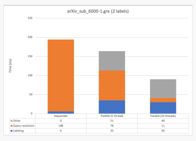
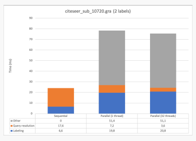
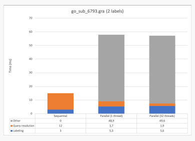
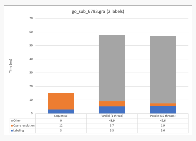
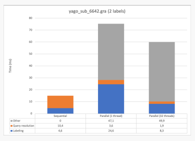
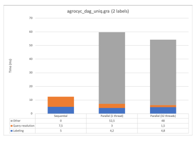
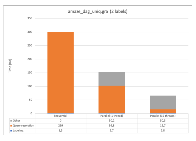
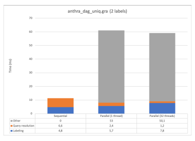
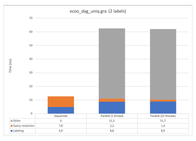
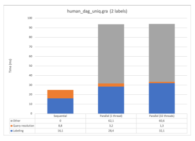
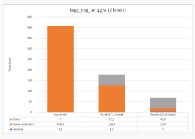
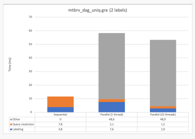
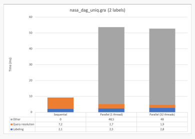
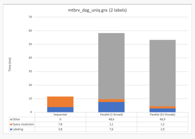
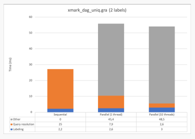
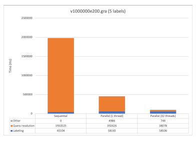
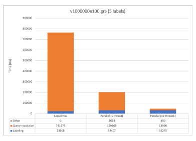
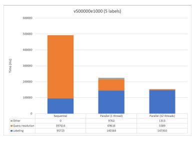
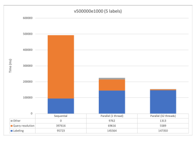
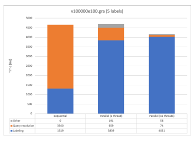
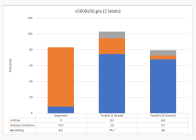
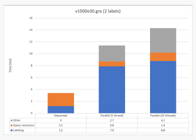
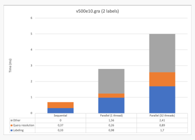
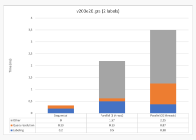
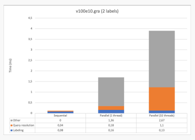
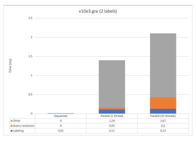
So if we have 2 vertex (v1, v2) and two queries, we will have something like:

- `el_query[0]` = Query 1 for V1 -> V2
- `el_query[1]` = Query 2 for V3 -> V4
- `el_query[0].num[0]` = V1 in Query 1
- `el_query[0].num[1]` = V2 in Query 1
- `el_query[0].can_reach` = true/false based on V1 -> V2

Time and Memory usage

Hardware configuration:

- CPUs: 2x 8-core Xeon E5-2690 (32 threads)
- RAM: 32GB 1333MHz DDR3 ECC

DAG Category					
LARGE REAL					
SMALL DENSE REAL					
SMALL SPARSE REAL					
continues					
SYNTHETIC LARGE					
SYNTHETIC SMALL					
continues					

From the histograms above, we can see that in most cases our program (the parallel implementation) achieves comparable or better results than its sequential counterpart. In very small DAGs, we can see that the overhead due to the creation of the threads and their data structures overcomes the advantage of the parallelization. But, as the DAGs grow in number of nodes and edges, we can see that the parallelization offers more and more a great improvement in time usage (e.g. with v1000000e200 the parallel version takes only 4.8% of the time taken by the sequential version).

Note: the sequential version does not keep track of the file read times (denoted as "Other" in gray for the parallel version). We can assume a similar or worse time with respect to the parallel version, so we can mostly safely ignore the gray part of the bar of the parallel version.

Memory (MiB) / Real DAG	cit-Patents.scc	uniprotenc_22m.scc	uniprotenc_100m.scc	arXiv_sub_6000-1	citeseer_sub_10720	go_sub_6793	pubmed_sub_9000-1	yago_sub_6642
Sequential	435	178	1761	2.7	2.9	2.6	2.8	2.6
Parallel (32 threads)	940	305	3036	5.5	6.6	5.4	6	5.6

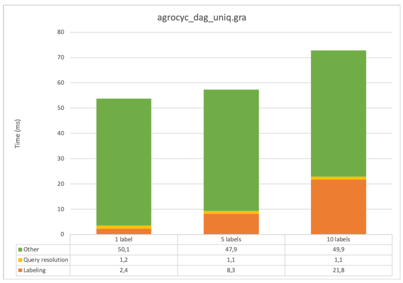
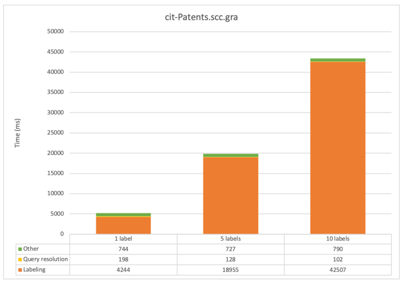
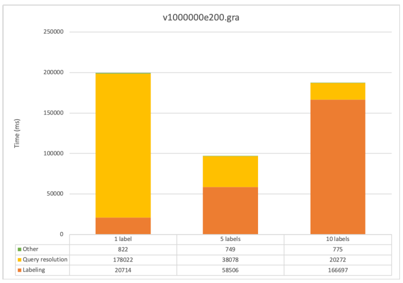
Memory (MiB) / Real DAG	agrocyc_dag_uniq	amaze_dag_uniq	anthra_dag_uniq	ecoo_dag_uniq	human_dag_uniq	kegg_dag_uniq	mtbrv_dag_uniq	nasa_dag_uniq	vchocyc_dag_uniq	xmark_dag_uniq
Sequential	3	2.2	3	3	5.5	2.2	2.8	2.5	2.7	2.5
Parallel (32 threads)	5.9	4.5	5.8	5.8	9.4	4.6	5.4	5	5.4	5.2

Memory (MiB) / Synthetic DAG	v1000000e100	v1000000e200	v100000e100	v500000e1000	v500000e50	v1000e30	v100e10	v10e3	v200e20	v5000e50	v500e10
Sequential	308	542	32	1180	96	1.6	1.5	1.5	1.5	2.4	1.5
Parallel (32 threads)	467	659	48	1093	187	2.8	2.5	2.1	2.5	4.3	2.6

From the tables above, it emerges that in our parallel version we almost always use more memory than the sequential version. This is most likely due to the additional data structures we need to store thread arguments, indexes, roots, mutexes and barriers.

Note: the results above have been obtained using the optimal number of labels (2 for small DAGs, 5 for large DAGs, as per the GRAIL paper).
Note: to reproduce the results stored in the `logs` directory for the parallel version, simply run `./complete_benchmark.sh`.

A brief analysis of the best number of labels

Small DAG	Large Real DAG	Large Synthetic DAG
		

	1 label	5 labels	10 labels
🟩 Other	50,1	47,9	49,9
🟨 Query resolution	1,2	1,1	1,1
🟧 Labeling	2,4	8,3	21,8

	1 label	5 labels	10 labels
🟩 Other	744	727	790
🟨 Query resolution	198	128	102
🟧 Labeling	4244	18955	42507

	1 label	5 labels	10 labels
🟩 Other	822	749	775
🟨 Query resolution	178022	38078	20272
🟧 Labeling	20714	58506	166697

Having run these tests with 32 threads and 1, 5, and 10 labels, we can see that the best number of labels in terms of time differs based on the dimension of the DAG:

- if the DAG is small, we should prefer a small number of labels, such as 1 or 2
- if the DAG is large, if every vertex has many edges the best number of labels is around 5

These results confirm the contents of the GRAIL paper.

How to run

We have created a Bash wrapper (`run.sh`) around our C program that allows for a quick execution of all of its parts and provides some added functionalities.

```
Use -h or --help to show this help.
Usage:
./run.sh ([-d|--download] [-g|--generate] [-r|--run [-l|--labels <number of labels>] [-t|--threads <number of threads>] [benchmark|test|<path to graph file>]])
[] mean an argument is optional
() mean an argument is mandatory
| means the argument on its left is equivalent to the argument on its right
It is required to give at least 1 argument to let the script know what is expected of it.

There are 4 run modes available in this script:
- 'specific': if you give a path to a .gra file, it will run our program against it
- 'test': it will run our program against all the generated DAG files in data/gen-dags (it is required to run ./run.sh -g|--generate at least once in order to use this mode)
- 'benchmark': it will run our program against all the downloaded DAG files in data/grail-dags (it is required to run ./run.sh -d|--download at least once in order to use this mode)
- 'all': it will run our program against all the DAGs in data/gen-dags and data/grail-dags. It is equivalent to running both modes above.
The default run mode is 'all'.
The default number of labels is 5, but it can be specified with the -l|--labels <num> option.
The default number of threads is the maximum number of threads of your CPU.

You need the following packages to run this script:
wget gzip tar make gcc
```

So if you want to download all the needed DAG files that come with the GRAIL repository, you can simply use:
`./run.sh --download`
If you want to generate all the DAG files with prof. Quer's code, use:
`./run.sh --generate`
If you want to run our program against a single DAG file:

`./run.sh --run --labels 2 data/gen-dags/v100e10.gra`
If you want to run our program against all benchmark (GRAIL) DAG files:
`./run.sh --run benchmark`
If you want to do everything in one line:
`./run.sh --download --generate --run --labels 4 all`.

See the help message above for all the other options.

Note: this script always re-compiles the code if `make` detects a change.