Politecnico di Torino
Master’s Degree in Computer Engineering

SDP – System and Device Programming

Project 2: Reachability Queries in Directed Graphs

Professor: Stefano Quer
Students: Germinario Federica, Pisani Alessandro
Academic year: 2019/2020

Project 2: Reachability Queries in Directed Graphs

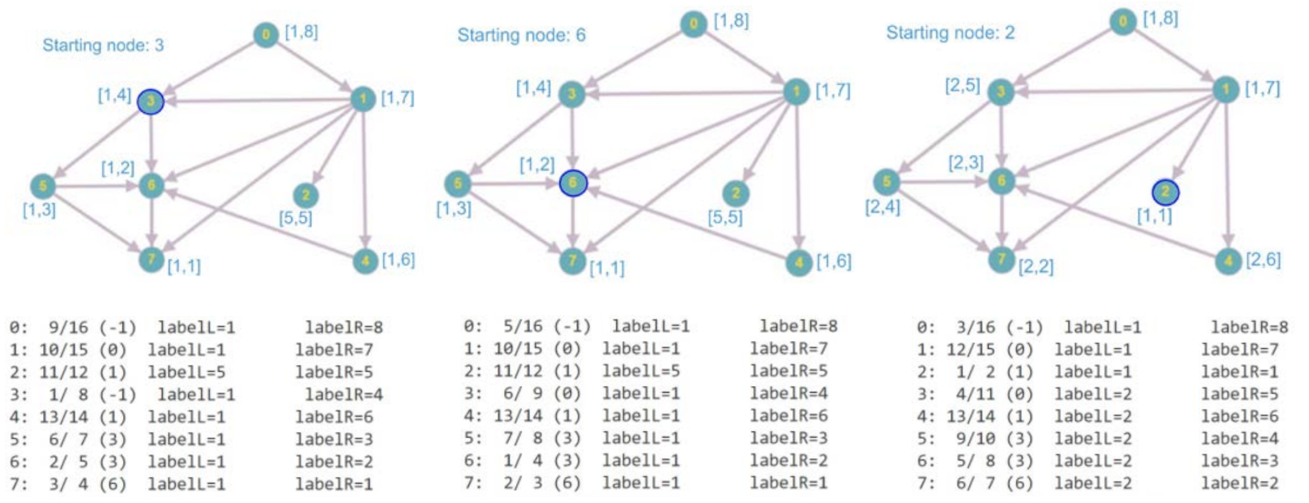
1. INTRODUCTION

In this project we worked with directed acyclic graphs $G = (V, E)$, where V is the set of vertices and E is the set of directed edges. A *reachability query* asks if there exists a path from u to v in G: If such a path exists, we say that u can reach v ($u \rightarrow v$), otherwise we say that u cannot reach v ($u \nrightarrow v$).

GRAIL, which stands for **G**raph **R**eachability Indexing via R**A**ndomized **I**nterval **L**abeling, instead of using a single interval, as previous methods did, employs multiple intervals that are obtained via random graph traversals. We use the symbol d to denote the number of intervals to keep per node, which also corresponds to the number of graph traversals used to obtain the label.

The key idea of GRAIL is to do very fast elimination for those pairs of query nodes for whom non-reachability can be determined via the intervals. In other words, if $L_v \nsubseteq L_u$, which can be checked in $O(d)$ time, we immediately return $u \nrightarrow v$. On the other hand, if successive index lookups fail, reachability defaults to a DFS in the worst-case.

In the figure there is an example in which a multiple interval labeling is employed.



```
0:  9/16 (-1)  labelL=1        labelR=8
1: 10/15 (0)  labelL=1        labelR=7
2: 11/12 (1)  labelL=5        labelR=5
3:  1/ 8 (-1)  labelL=1        labelR=4
4: 13/14 (1)  labelL=1        labelR=6
5:  6/ 7 (3)  labelL=1        labelR=3
6:  2/ 5 (3)  labelL=1        labelR=2
7:  3/ 4 (6)  labelL=1        labelR=1
```

```
0:  5/16 (-1)  labelL=1        labelR=8
1: 10/15 (0)  labelL=1        labelR=7
2: 11/12 (1)  labelL=5        labelR=5
3:  6/ 9 (0)  labelL=1        labelR=4
4: 13/14 (1)  labelL=1        labelR=6
5:  7/ 8 (3)  labelL=1        labelR=3
6:  1/ 4 (3)  labelL=1        labelR=2
7:  2/ 3 (6)  labelL=1        labelR=1
```

```
0:  3/16 (-1)  labelL=1        labelR=8
1: 12/15 (0)  labelL=1        labelR=7
2:  1/ 2 (1)  labelL=1        labelR=1
3:  4/11 (0)  labelL=2        labelR=5
4: 13/14 (1)  labelL=2        labelR=6
5:  9/10 (3)  labelL=2        labelR=4
6:  5/ 8 (3)  labelL=2        labelR=3
7:  6/ 7 (6)  labelL=2        labelR=2
```

Figure 1: Multiple Interval Labeling

From this example we can make some considerations:

- Label pairs that obtained starting from node 3 are equal to the one we obtain starting from node 6
- Looking at the labels obtained from the DFS starting at node 3 it seems that node 4 can reach node 2 ($4 \rightarrow 2$), since $[5,5] \subseteq [1,6]$. But if we check the same condition from labels calculated with the DFS starting from node 2 we have that $[1,1] \nsubseteq [2,6]$

From this considerations we can understand the importance of adding more label pairs to each node in order to have correct analysis of queries.

The space complexity of GRAIL indexing is $O(d \cdot n)$, since d intervals have to be kept per node, and the construction time is $O(d(n + m))$, since d random graph traversals are made to obtain those labels. Since d is typically a small constant, GRAIL requires time and space linear in the graph size for index creation.

For query answering, the time complexity ranges from $O(d)$ (in cases where non-reachability can be determined using the index) to $O(n + m)$ (in cases where the search defaults to a DFS).

2. THE GRAIL APPROACH

The motivating idea in GRAIL is to use interval labeling multiple times to reduce the workload of the second phase of indexing or the querying.

In GRAIL, for a given node u, the new label is given as $L_u = L_u^1, L_u^2, \dots, L_u^d$, where L_u^i is the interval label obtained from the i-th (random) traversal of the DAG, and $1 \le i \le d$, where d is the dimension or number of intervals.
We say that L_v is contained in L_u, denoted as $L_v \subseteq L_u$, if and only if $L_v^i \subseteq L_u^i$ for all $i \in [1, d]$.

If $L_v \nsubseteq L_u$, then we can conclude that $u \nrightarrow v$, as per the theorem below:

THEOREM 1. If $L_v \nsubseteq L_u$, then $u \nrightarrow v$.[1]

On the other hand, if $L_v \subseteq L_u$, it is possible that this is a false positive, i.e., it can still happen that $u \nrightarrow v$. We call such a false positive containment an *exception*. In general using multiple intervals drastically cuts down on the exception list, but is not guaranteed to completely eliminate exceptions.

There are two main issues in GRAIL:

1) how to compute the d random interval labels while indexing, and
2) how to deal with exceptions, while querying.

We discuss these issues in detail below.

2.1. Reachability Index Construction

The core idea behind the GRAIL approach lies on the observations that for direct trees (DTs) it is possible to build an index that occupies linear space, takes linear construction time and is able to answer reachability queries in constant time, and that a DT is a special case of a DAG where every vertex has a single parent or, conversely, that a DAG can be seen as a set of overlapping DTs.

In fact, for direct trees it is possible to build an index using the **min-post labeling** technique, which assigns to each node u an interval label L_u such that $L_u = [s_u, e_u]$ where

- e_u, or outer rank, is defined as the rank of node u in a post order traversal and
- s_u, the inner rank, is the minimum e_u among the descendants of u.

$$\forall u \in V, L_u = [s_u, e_u] \; s.t. \; s_u = min\{s_x \mid s_x \in descendants(u)\}, e_u = postorder(u)$$

The fact that in direct trees every node has a single parent has strong implications for reachability queries, indeed it is possible to determine whether any vertex reaches any other node in constant time by interval

[1] For the Proof of THEOREM 1 see the original paper, pag 278.

containment. This is due to the fact that if node u has a bigger outer rank value than v's, it will be visited later in a post-order traversal, whereas having the same inner rank implies that v is one of u's descendants.

Min-Post Labeling in function `graph_dfs_r()`

```
n->right_label[index]= ++post_order_index[index];
n->left_label[index]=g->nv + 1;

if(n->head==NULL)
    n->left_label[index]=n->right_label[index];
else{
    for(e=n->head; e!=NULL; e=e->next){
        t = e->dst;
                if(t->left_label[index]<n->left_label[index])
                    n->left_label[index]=t->left_label[index];
        }
    }
```

In function `graph_dfs_r` we created the right and left labels for each node of the graph.

The right label corresponds to e_u and the rank is derived from the post order traversal performed by the current thread. The left label s_u is firstly initialized to a maximum which is represented by the number of vertices and then gets assigned to the minimum e_u among the descendants of node u.

We generated the desired number of post-order interval labels by simply changing the visitation order of the children randomly during each DFT.

In terms of the traversal strategies, we aim to generate labelings that are as different from each other as possible. The GRAIL approach experimented with three different traversal strategies: Randomized, Randomized Pairs and Bottom-Up. In GRAIL implementation the adopted one is the Randomized strategy.

2.2 Reachability Queries

To answer reachability queries between two nodes, u and v, GRAIL adopts the following approach:

1) If $L_v \nsubseteq L_u$ then we can immediately conclude that $u \nrightarrow v$.
2) Else if $L_v \subseteq L_u$, then nothing can be concluded immediately since we know that the index can have false positives.

There are basically two ways of tackling exceptions. The first is to explicitly maintain an exception list per node. Unfortunately, keeping explicit exception lists per node adds significant overhead in terms of time and space, and further does not scale to very large graphs. Thus GRAIL adopts a second approach which uses a "smart" DFS, with recursive containment check based pruning, to answer queries. This strategy does not require the computation of exception list so its construction time and index size are linear.

GRAIL Query: Reachability Testing

```
int isContained(vertex_t *u, vertex_t *v, int d){
    for(int i=0; i<d; i++){
        if(v->left_label[i] < u->left_label[i] ||
                            v->right_label[i] > u->right_label[i])
            return 0;
    }
    return 1;
}
```

```
int isReachableDFS(vertex_t *u, vertex_t *v, graph_t *g, int d){
    if(!isContained(u, v, d))
        return 0;
    else if(u->id == v->id)
        return 1;
    else{
        edge_t *e;
        vertex_t *t;
        e = u->head;
        while (e != NULL) {
            t = e->dst;
                t->pred = u;
                    if(isReachableDFS(t, v, g, d))  // default recursive DFS with pruning
                        return 1;
            e = e->next;
        }
        return 0;
    }
}
```

This algorithm shows the code for reachability testing in GRAIL. We test whether $L_v \not\subseteq L_u$, and if so, returns `false`. Following we have the default recursive DFS with pruning. If there exists a child c of u, that satisfies the condition that $L_v \subseteq L_u$, and we check and find that $c \rightarrow v$, we can conclude that $u \rightarrow v$, and GRAIL returns `true`. Otherwise, if none of the children can reach v, then we conclude that $u \nrightarrow v$, and we return `false`.

3. MULTITHREADED GRAIL IMPLEMENTATION

3.1. Reachability Index Construction

In our parallel version we had to deal with the fact that multiple threads will need to perform a DFS on the same graph data structure. This implies that for each visited node we had to choose among two different strategies:

1) Consider graph vertexes as critical sections protected with mutexes or binary semaphores
2) Create an array, of length equal to the number of indexing labels, for each vertex data structure involved in the DFS.

So that each thread can work in parallel independently from the others in order to accomplish a clean multithreaded implementation of DFS.

As an example our version of the DFS uses colors (WHITE, GREY, BLACK[2]). In particular the colouring strategy allows to identify graph islands, to find them after the DFS has been performed we iterate over the vertexes of the graph and if a node is white we call the DFS from that node. In order to handle this case we created for each node an array of colors as long as the number of indexing labels so that each thread can work and complete the DFS without mistakes.

Following what has been done for the colors we have defined for each vertex a couple of arrays, of length equal to the number of labels, for the left and right labels.

[2] The colors respectively indicate if a node has not been visited yet, if a node has been visited but not all of its children have been visited and if a node has been visited completely.

The reason behind this choice is that the number of indexes d for each vertex is bounded between 3 and 5^3, it implies an increase in memory which is proportional with the number of labels.

With the following code we generated the desired number of post-order interval labels by simply changing the visitation order of the children randomly before each DFS. Each Labeling is performed by a thread.

Randomized Labeling in `main()`

```
for(int j=0;j<labelNum;j++){
    // Randomized traversal strategy (RandomizedLabeling)
    do{
        i = Randoms(lower, g->nv-1, count);
        src= graph_find(g, i);
    }while(src->visited!=-1);

    src->visited = 1;     // to avoid selecting the same root for DFS
    ...
    pthread_create(&td[j].threadID, NULL, graph_dfs, (void *) &td[j]);
}
```

3.2. Reachability Queries

We implemented two multithreaded versions to deal with reachability queries.

Version 1 – Parallel File Reading

In order to parallelize the Reachability testing we allowed threads to read concurrently from the file containing the couples of queries. When we need to allow threads to read concurrently from a file two approaches can be used:

1) Have a Critical section for file reading protected by a semaphore or a mutex. In this way we guarantee mutual exclusion.
2) Divide the file by the number of threads, each thread will be assigned with its own part of the file.

MULTITHREADED Query File Reading

```
do{
        pthread_mutex_lock(&sem);
            readval = fscanf(td->fp, "%d %d", &tmp1, &tmp2);  //CS
        pthread_mutex_unlock(&sem);
        if(readval != EOF){
            src= graph_find(g, tmp1);
            dst= graph_find(g, tmp2);
            if(isReachableDFS(src, dst, g, labelNum))
                    printf("%d reaches %d\n", tmp1, tmp2);
            else
                    printf("%d does not reach %d\n", tmp1, tmp2);
        }
}while(readval != EOF);
```

The advantage of the first approach is that we don't need to know the number of lines of the file, while in the second case we need it. Moreover, if we have a slower thread, in the first approach faster threads will make the issue not noticeable. In the second case instead a slower thread would impact performances. Thus we opted for the first approach.

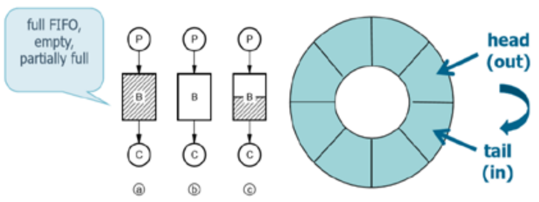
[3] Increasing the number of intervals increases construction time, but yields decreasing query times. However, increasing d does not continue to decrease query times, since at some point the overhead of checking a larger number of intervals negates the potential reduction in exceptions. In GRAIL experiments, they found out that the best query time is obtained when d = 5 or smaller (when the average degree is smaller).

Since the semaphore being used was a binary semaphore, we opted for mutexes from pthread library since they are faster and lighter than semaphores for this specific situation.

Version 2 – Producer and Consumer scheme

It must be taken in consideration that, generally speaking, is not a good idea to parallelize disk I/O. Hard disks do not like random I/O because they have to continuously seek around to get to the data. Assuming you're not using RAID, and you're using hard drives as opposed to some solid state memory, you will see a severe performance degradation if you parallelize I/O (even when using technologies like those, you can still see some performance degradation when doing lots of random I/O).

This is why we implemented another approach based on the *Producer and Consumer* Paradigm which uses a circular buffer of dimension **SIZE** to store the elements to be produced and consumed. The circular buffer implements a FIFO queue



With 1 Producer and C Consumers we allow the operations enqueue and dequeue on different extremes of the queue and at the same time also concurrent access operations to the same extreme of the queue are possible.

To count full and empty elements in the queue:

- A semaphore "full" counts the number of filled elements
- A semaphore "empty" counts the number of empty elements

This enforces **Mutual exclusion** among producers and among consumers, as they act on opposite extremes of the buffer

- Producers and consumers can work concurrently
- As long as the queue is not completely full or completely empty

MULTITHREADED Query File Reading: Data Structures

```
struct query_s{
    int src;
    int dst;
};
#define SIZE 1024
//FIFO standard non ADT without n
query_t queue[SIZE];  int head=0, tail=0;
sem_t empty;
sem_t full;
...
pthread_mutex_init(&sem, NULL);
sem_init(&full, 0, 0);
sem_init(&empty, 0, SIZE);
```

```
void enqueue(query_t query){
    queue[tail] = query;
    // queue[tail].dst = query.dst;
    // queue[tail].src=query.src;
    tail=(tail+1)%SIZE;
    return;
}
void dequeue(query_t *query){
    *query = queue[head];
    head=(head+1)%SIZE;
    return;
}
```

In this way we allowed:

- One Main Producer thread which takes care of reading the file and of storing the queries read from it inside the circular queue.

- C Consumer threads which read from the circular buffer and then execute the sequential reachability test. These threads must be synchronized with the producer thread in order to read only when the buffer is not completely empty and between themselves in order to avoid accessing the same element in the same index of the buffer. This is why we have another mutex which enables them to work in mutual exclusion.

MULTITHREADED Query File Reading: Producer

```
void queries_reader(char *filename, graph_t *g, int labelNum){
    query_t query;
    FILE *fp2= fopen(filename, "r");

    while(fscanf(fp2, "%d %d", &query.src, &query.dst)!=EOF){
        sem_wait(&empty);
            enqueue(query);
        sem_post(&full);
    }

    fclose(fp2);
    return;
}
```

MULTITHREADED Query File Reading: Consumer

```
void *queries_checker(void *param){
    threadD *td ;
    td = (threadD *) param;
    graph_t *g = td->g; int labelNum = td->labelNum;
    vertex_t *src, *dst; query_t query;

    do{
        sem_wait(&full);
        pthread_mutex_lock(&sem);
            dequeue(&query);
        pthread_mutex_unlock(&sem);
        sem_post(&empty);
        // CONSUME
            src= graph_find(g, query.src);
            dst= graph_find(g, query.dst);
            if(isReachableDFS(src, dst, g, labelNum)){
                    printf("%d reaches %d\n", query.src, query.dst);
    }while(1);

    return (int *)1;
}
```

4. CONCLUSIONS

We conducted experiments to compare our parallel implementation of the sequential algorithm. All experiments are performed in a machine x86_64 Intel® Core™ i5-9400F CPU @ 2.90GHz GNU/Linux which has 6 processors and 32G RAM.

We have seen a positive outcome from our parallel implementation of the algorithm. Our experiments are reported only with randomized traversals and DFS with pruning. We generated 100K random query pairs, and issue the same queries to all graphs for all the possible combinations of label number chosen.

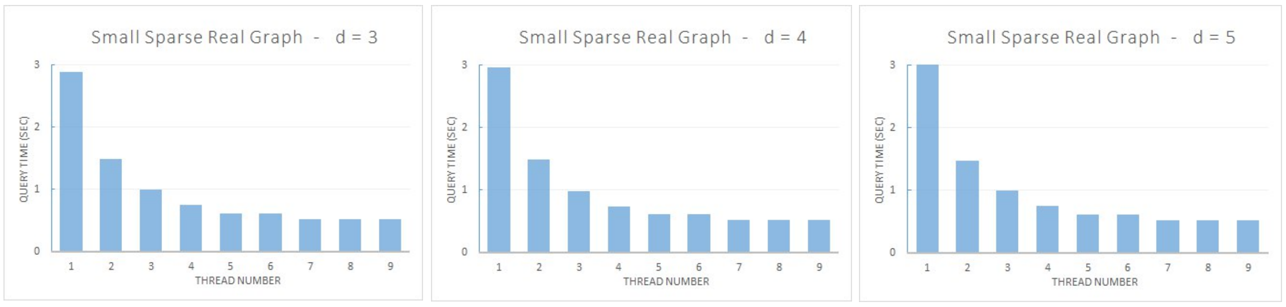
In the case of the Parallel version we issued the same queries also for all the possible combinations against increasing number of threads until we found out, as expected, that above a certain threshold, increasing the number of threads was no more efficient but instead leads to degradation of performances.

In this report we show the outcome of our experiments with particular attention to `xmark` and `citeseer` datasets, respectively, a *small sparse real* and a *small dense real* dataset.

Dataset	**Nodes**	**Edges**	**Avg Deg**
xmark	6080	7051	1.16
citeseer	6000	66707	11.12

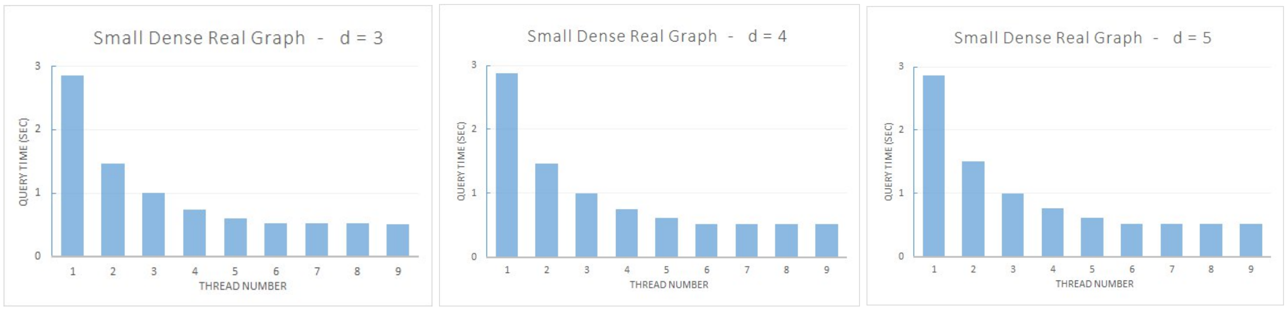
4.1 Time Efficiency

In Figure 3 we plot, for the query time the effect of (i) increasing the dimensionality of the index and (ii) increasing the number of threads.



(a) (b) (c)

Figure 2: Effect of Increasing Number of Intervals on `xmark`



(a) (b) (c)

Figure 3: Effect of Increasing Number of Intervals on `citeseer`

	Xmark - sparse			**Citeseer - dense**		
#Threads	**Query Time** (sec)					
	d=3	**d=4**	**d=5**	**d=3**	**d=4**	**d=5**
1	2.883085	2.962273	3.010052	2.853750	2.884290	2.860379
2	1.491768	1.491194	1.479550	1.471226	1.458576	1.501348
3	0.995653	0.980823	0.996801	1.009363	0.990616	0.997924
4	0.751908	0.743054	0.752142	0.749265	0.756383	0.757906
5	0.614458	0.611443	0.611483	0.609090	0.610191	0.605981
6	0.611702	0.610530	0.611950	0.521270	0.519718	0.513292
7	0.518061	0.518726	0.521073	0.519268	0.518164	0.521863
8	0.525958	0.524964	0.523864	0.523116	0.517775	0.519419
9	0.520364	0.521890	0.524543	0.515804	0.514418	0.519206

In figure 4 it is possible to see how the parallel version leads to a huge improvement in terms of query time with respect to the sequential one.

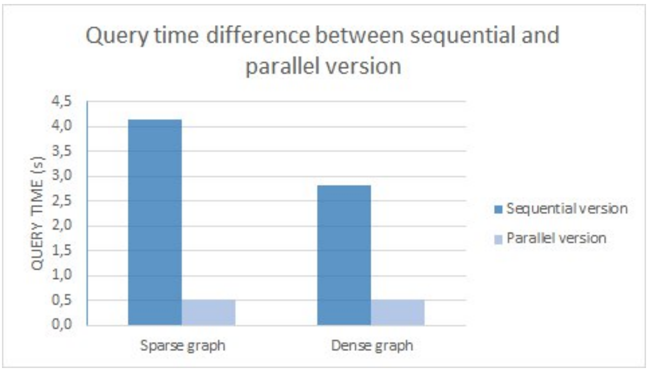


Figure 4: Query time difference on `xmarq`, `citeseer` for d=4

`Xmark` - sparse		`Citeseer` - dense	
Sequential version	Parallel version #threads = 7	Sequential version	Parallel version #threads = 7
4.130012	0.518726	2.826050	0.519268

Figure 5: Query time difference on `xmark`, `citeseer` for d=4

To compute how long it takes to run various versions of our programs so that we can determine if adding additional threads to our computation is worth it we used Using OpenMP functions for timing code.
OMP library function ‘signature’ looks like this:

```
#include <omp.h>

double omp_get_wtime( void );
```

4.2 Memory Efficiency

To check memory efficiency we used **Valgrind**, a framework that provides instrumentation to user-space binaries. It ships with a number of tools that can be used to profile and analyze program performances and provide analysis that can aid in the detection of memory errors such as the use of uninitialized memory and improper allocation or deallocation of memory. All are included in the valgrind package, and can be run with the following command:

```
valgrind --tool=toolname program
```

Memcheck is the default Valgrind tool, and can be run with valgrind *program*, without specifying `--tool=memcheck`. It detects and reports on a number of memory errors that can be difficult to detect and diagnose, such as memory access that should not occur, the use of undefined or uninitialized values, incorrectly freed heap memory, overlapping pointers, and memory leaks.

However, one must be aware that Valgrind's instrumentation will cause the program to run more slowly than it would normally.

We report below the memory efficiency of our sequential and parallel implementations of the program for the real sparse graph `xmark`.

4.2.1 Sequential version

```
==604== HEAP SUMMARY:
==604==     in use at exit: 0 bytes in 0 blocks
==604==   total heap usage: 31,379 allocs, 31,379 frees, 885,876 bytes allocated
==604==
==604== All heap blocks were freed -- no leaks are possible
==604==
==604== For counts of detected and suppressed errors, rerun with: -v
==604== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

4.2.2 Parallel version I

```
==692== HEAP SUMMARY:
==692==     in use at exit: 1,646 bytes in 5 blocks
==692==   total heap usage: 31,396 allocs, 31,391 frees, 923,138 bytes allocated
==692==
==692== LEAK SUMMARY:
==692==    definitely lost: 0 bytes in 0 blocks
==692==    indirectly lost: 0 bytes in 0 blocks
==692==      possibly lost: 0 bytes in 0 blocks
==692==    still reachable: 1,646 bytes in 5 blocks
==692==         suppressed: 0 bytes in 0 blocks
==692== Rerun with --leak-check=full to see details of leaked memory
==692==
==692== For counts of detected and suppressed errors, rerun with: -v
==692== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

4.2.3 Parallel version II

```
==340== HEAP SUMMARY:
==340==     in use at exit: 1,646 bytes in 5 blocks
==340==   total heap usage: 31,393 allocs, 31,388 frees, 873,514 bytes allocated
==340==
==340== LEAK SUMMARY:
==340==    definitely lost: 0 bytes in 0 blocks
==340==    indirectly lost: 0 bytes in 0 blocks
==340==      possibly lost: 0 bytes in 0 blocks
==340==    still reachable: 1,646 bytes in 5 blocks
==340==         suppressed: 0 bytes in 0 blocks
==340== Rerun with --leak-check=full to see details of leaked memory
==340==
==340== For counts of detected and suppressed errors, rerun with: -v
==340== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5. REFERENCES

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.174.5807&rep=rep1&type=pdf

http://selkie.macalester.edu/csinparallel/modules/MulticoreProgramming/build/html/timingAndScalability/TimingOnMTLandscalability.html

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-valgrind