Epidemic Simulation

Manos Chatzakis
csd4238@csd.uoc.gr

April 2021

1. Introduction

This report is part of the second assignment of cs342 - Parallel Programming course of Computer Science Department, University of Crete. It aims to simulate the spread of a virus on a static graph representing the people and to parallelize the calculations using openMP. The report is organized as follows: Section 1 is the introduction. Section 2 describes the input graph. Section 3 analyzes the parallelism method. Section 4 shows the measurements. Section 5 is the conclusion.

2. Input Graph

The input is a undirected graph representing the connections of the people.

```
struct Graph {
    long maxSize;
    long currSize;
    Node *nodes;
};
```

```
struct Connection {
    long indexTo;
    short contaminates;
    struct Connection *next;
};
```

```
struct Node {
    long id;
    short isDead
    short isContaminated;
    short hasAnosia;
    short daysRecovering;
    long connections;
    Connection *connectionsHead;
};
```

The graph is initialized using the input file (the initialization time is not concluded to the time calculations), and the starting patients are loaded by a seed file, containing their IDs.

3. Implementation

In this section the implementation of the algorithm and the parallelism method are described.

3.1 Algorithm

Every day of the simulation consist of two phases. Phase one is the phase that the new patients are determined, by checking if a healthy person has a contaminated connection. During the second phase, the new patients from phase one are recorded, while the statistics of the current day are gathered and calculated. This is repeated every day.

3.2 Parallelism

The aim is to develop a parallel algorithm to run the above algorithm concurrently, without the use of locks. The following parallelism method requires only a barrier between the two phases: The threads are assigned chunks of graph nodes to calculate, using the default OpenMP split. For the first phase we care only for alive nodes that have not been contaminated yet. For these nodes the neighbour list is scanned. If the connection is going to contaminate the current node also, we set the corresponding flag in the list struct, to indicate that this node will be contaminated in the next day. Note that as the only think we write in the memory is the struct field which only the thread of the current node has access, there is no data race:

```
/* Phase 1 */
#pragma omp for firstprivate(seed, case_found) private(j, conn) schedule(static) \
                                                                   reduction(+: active)
for (j = 0; j < g->currSize; j++) {
    /*
        For every no-yet contaminated, alive node,
        we check if he will get contaminated during the day
    */
    if (!nodes[j].isDead && !nodes[j].hasAnosia && !nodes[j].isContaminated) {

        /* Iterating the connection list to find out if any neighbor is contaminated */
        conn = nodes[j].connectionsHead;
        while (conn) {
            if (!nodes[conn->indexTo].isDead &&
                nodes[conn->indexTo].isContaminated &&
                isGoingToContaminate(seed)){
                /* In this case, node[j] gets contaminated */
                case_found = 1;         /* Flag to add up active cases */
                /* This memory address is written only by the current thread, no data race*/
                conn->contaminates = 1;
            }
            conn = conn->next;
        }

        /* If node[j] gets contaminated, the active cases are increased */
        if (case_found) {
            active++;
            case_found = 0;
        }
    }
} /* Implicit (required) barrier between the two phases */
```

It is clear that during the first phase, the data of the nodes containing the state of the epidemic are just read, not written, thus the loop can be paralleled without any dependencies among the different threads.
Considering the second phase, we only care for each node individually. For every contaminated node, we calculate the probability to die during the current infection day. If the corresponding person survives, we increment the recovering days counter and check if the duration has passed in order to note him as an immune person. For every not contaminated node, his connection list is traversed in order to check the flags that were set during the previous phase, and the actions if it comes out that the person got contaminated are done accordingly:

```
/* Phase 2 */
#pragma omp for schedule(static) private(j, conn) firstprivate(seed)      \
    reduction(+                                                           \
            : totalDeaths, newDeaths, recovered, newCases, totalCases) \
        reduction(-                                                       \
                : active)
for (j = 0; j < g->currSize; j++) {
    /* We dont care for immune or dead nodes */
    if (nodes[j].hasAnosia || nodes[j].isDead) { continue; }

    /* Actions for contaminated nodes */
    if (nodes[j].isContaminated) {

        /* If the current node is going to die, the corresponding counters are updated */
        if (isGoingToDie(nodes[j].daysRecovering, seed)) {
            nodes[j].isDead = 1;
            totalDeaths++;
            newDeaths++;
```

```
                active--;
            }
            else {
                /* If the current node survives this day, the counter is incremented */
                nodes[j].daysRecovering++;
                /*
                    In case the virus duration has passed,
                    the current node survives and becomes immune
                */
                if (nodes[j].daysRecovering == DURATION) {
                        nodes[j].isContaminated = 0;
                        nodes[j].hasAnosia = 1; /* Immune nodes cannot get ill again */
                        recovered++;
                        active--;
                }
            }
        }

        /* Actions for not contaminated nodes */
        else {
                /*
                    Traversing the neighbour list to find out
                    if the current node got contaminated during phase 1
                */
                conn = nodes[j].connectionsHead;
                while (conn) {
                    /* Reading the data written in phase 1 */
                    if (conn->contaminates == 1) {
                        nodes[j].isContaminated = 1; /* Contamination Case */
                    }

                    conn->contaminates = 0; /* Reseting the flag for the next days */
                    conn = conn->next;
                }

                /* If the node got contaminted, increase the counters */
                if (nodes[j].isContaminated) {
                    newCases++;
                    totalCases++;
                }
        }
} /* Implicit (required) barrier */
```

It is clear that the above code does not result in any data race, as it processes each node independently.

3.3 Load Balancing

As the graph represents people connections, it is possible that the openMP chunk division is not the most fair, as different threads can be assigned different amount of work. Given that graph partition/reordering to shuffle the nodes in a way that high work nodes are assigned to different threads is an NP problem and could result in high overhead, the mechanisms of openMP were used: The program was tried with different scheduling configurations, like static, dynamic, guided and auto. Among these, the best running time was for static scheduling. Other scheduling policies resulted in high overhead in order to set up and process the scheduling environment.

4. Measurements

In this section, the measurements are presented, in order to test the efficiency. For these measurements, the SNAP data-sets where used. The calculations were done using an automated python script. Note that the running times contain a small constant overhead, as the results are written to the output file concurrently with the calculations and there are other initializations (custom seed set up etc.) before the actual parallel code is executed:

```
clock_gettime(CLOCK_MONOTONIC, &start);
epidemic(zeroPatients, threads, days, outstream);
clock_gettime(CLOCK_MONOTONIC, &finish);
```

4.1 Speedup, Throughput and Standard Deviation

For speedup, two versions of the calculations are used, SpeedUp(C) and SpeedUp(N). SpeedUp(C) is the speedup in comparison with the sequential code (compiled without -openmp), while SpeedUp(N) is the speedup between thread N and N-1 (etc. the speedup between thread 2 and thread 4):

$$SpeedUp(C) = \frac{t_{seq}}{t_{par}} = \frac{Sequential_{average}}{threadN_{average}} = C$$

$$SpeedUp(N) = \frac{t_{seq}}{t_{par}} = \frac{threadN-1_{average}}{threadN_{average}} = N$$

Throughput is the metric that shows the computation capabilities of the program among the total days and selected threads, as it is calculated using the formula:

$$Throughput = \frac{TotalSimulatedDays}{t_{average}} = Y\frac{days}{seconds}$$

Standard Deviation is calculated for each configuration and input size using the equation:

$$Stdev = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i-\bar{x})^2} = V seconds$$

4.1.1 Facebook Dataset

- 30 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.013766	0.012913	0.011802	0.013536	0.011493	0.01270	-	-	2362.2	0.00102
1	0.012189	0.014579	0.014579	0.012525	0.012312	0.01289	-	-	2327.3	0.00098
2	0.009211	0.00781	0.00731	0.007334	0.009072	0.00815	1.55x	1.55x	3680.9	0.00093
4	0.004491	0.004681	0.004769	0.004765	0.004544	0.00465	2.73x	1.75x	6451.6	0.00013

- 180 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.015799	0.016437	0.016823	0.015517	0.016417	0.01620	-	-	11111.1	0.00053
1	0.015805	0.016862	0.016112	0.016324	0.015635	0.01615	-	-	11145.5	0.00048
2	0.009733	0.010075	0.009756	0.009865	0.009878	0.00986	1.64x	1.64x	18255.5	0.00014
4	0.006159	0.006169	0.006383	0.006269	0.006006	0.00620	2.61x	1.59x	29032.2	0.00014

- 365 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.020681	0.021928	0.021195	0.021766	0.020891	0.02129	-	-	17144.1	0.00054
1	0.021188	0.022447	0.0214	0.021594	0.021872	0.02170	-	-	16820.2	0.00049
2	0.012924	0.012791	0.012715	0.012832	0.01344	0.01294	1.64x	1.64x	28207.1	0.00029
4	0.00811	0.008302	0.008117	0.008193	0.007863	0.00812	2.62x	1.59x	44950.7	0.00016

4.1.2 Gnutella Dataset

- 30 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.029174	0.028695	0.02912	0.027951	0.028805	0.02875	-	-	1043.4	0.00049
1	0.027814	0.027059	0.02707	0.026215	0.027568	0.02715	-	-	1104.9	0.00061
2	0.018824	0.017921	0.018127	0.018582	0.018498	0.01839	1.56x	1.56x	1631.3	0.00036
4	0.012941	0.012948	0.012903	0.013297	0.013077	0.01303	2.20x	1.41x	2302.3	0.00016

- 180 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.05694	0.054214	0.055179	0.057482	0.054482	0.05566	-	-	3233.9	0.00147
1	0.05558	0.057051	0.052071	0.056956	0.056391	0.05561	-	-	3236.8	0.00206
2	0.03285	0.040195	0.032938	0.03245	0.032898	0.03427	1.62x	1.62x	5252.4	0.00332
4	0.024216	0.02435	0.020665	0.020613	0.020855	0.02214	2.51x	1.54	8130.0	0.00196

- 365 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.090817	0.091275	0.090647	0.090318	0.09094	0.2168	-	-	4013.6	0.00052
1	0.09138	0.092392	0.090254	0.090459	0.092121	0.09132	-	-	3996.9	0.00096
2	0.050682	0.050973	0.051027	0.051919	0.05134	0.05119	1.77x	1.77x	7130.2	0.00047
4	0.029888	0.030338	0.036835	0.030185	0.029792	0.03141	2.89x	1.62x	11620.5	0.00304

4.1.3 Enron Dataset

- 30 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.055284	0.054077	0.059461	0.056324	0.05147	0.05532	-	-	542.2	0.00294
1	0.057074	0.06082	0.055581	0.060135	0.061725	0.05907	-	-	507.8	0.00262
2	0.04372	0.048527	0.045012	0.045203	0.044275	0.04535	1.21x	1.21x	661.5	0.00187
4	0.038976	0.03621	0.036812	0.03568	0.037611	0.03706	1.49x	1.22x	809.4	0.00129

- 180 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.106174	0.117686	0.106538	0.106028	0.108394	0.10896	-	-	1651.9	0.00497
1	0.109633	0.122478	0.109258	0.10865	0.115489	0.11310	-	-	1591.5	0.00592
2	0.073844	0.082152	0.085098	0.073205	0.07521	0.07790	1.39x	1.39x	2310.6	0.00538
4	0.015658	0.015022	0.014813	0.014786	0.014616	0.05635	1.93x	1.38x	3194.3	0.00034

- 365 Days.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Speedup(C)	Speedup(N)	Throughput	Stdev
Sequential	0.149498	0.148297	0.165367	0.148535	0.147977	0.15193	-	-	2402.4	0.00753
1	0.151574	0.148704	0.151791	0.149653	0.152193	0.15078	-	-	2420.7	0.00152
2	0.099823	0.103231	0.101236	0.099969	0.106879	0.10223	1.48x	1.48x	3570.3	0.00294
4	0.078929	0.078061	0.080094	0.078359	0.078871	0.07886	1.92x	1.29x	4628.4	0.00078

4.2 Single Thread Parallelism Overhead of OpenMP

The data presented above show that although the sequential code and 1 thread openmp code have a very small time difference, the sequential code is little faster. This is normal, as openmp needs to set the multi-threaded environment when it's used. Also, when using openmp, no matter the number of threads, the implicit barriers at the end of the parallel regions and other library mechanisms add up to the total computation time. In general, OpenMP prepares the parallel regions for executions no matter how many threads are used.

4.3 Speedup Commenting

Regarding the number of days: From the tests that were done for the epidemic program, we can conclude the following result considering the days: As simulating the spread for more days requires bigger amount of work, multi-threaded implementation works faster in these cases. Given that the OpenMP environment has an overhead for its creation, using the threads to process more work (more days) could lead to better results.
Regarding the size of input graph: Generally, the speedup tends to become bigger for larger graphs, as the computation demands are higher. On the other hand, the size of the graph is not the only parameter to be considered. A crucial parameter regarding the speedup is the amount of work that every node has (load balancing parameter).