Documentation

SSSP Parallel

Andrej Loparev [557966]

Berlin, 24 VII 2020

Contents

List of Figures

List of Tables

List of Snippets

1 Introduction

This document describes the carried out project in the course Parallel Systems, during the summer semester 2020 at HTW Berlin. The purpose of the project is to deepen understanding of a chosen course topics by confronting one specific problem scenario and single handily developing a solution. The technology stack to be used consists of C++ and two parallel interfaces such as OpenMP, MPI, CUDA, OpenCL, and so on. The written code is to be managed by Git to display the development process in real-time, step by step. The solution in the form of an application shall be executable from the console and accept input parameters for initialization.

The chosen problem comes from the realm of graphs and finds the shortest path from a single point in the graph to all the others (SSSP). SSSP is fundamental in many application areas with transportation requirements. Moreover, it is not limited to physical forms only and can just as well be applied to network routing, for example. One possible solution provides the Dijkstra algorithm, which will be the major topic of this work. More precisely, we are interested if it can be parallelized effectively. The algorithm will be implemented sequentially in C++, to begin with. In addition, two distinct parallel versions shall be developed, one with OMP and another with CUDA. The Git repository that reflects the code history is freely accessible under https://github.com/aloparev/dijkstra.

2 Fundamentals

In this chapter, we are going to review the basics of parallel interfaces in brief, before proceeding to the fundamentals of Dijkstra and related concepts.

2.1 Open Multi-Processing

Open Multi-Processing, also known as OpenMP or simply OMP, is an applied programming interface used to create parallel applications, mostly for parallelized computing in shared memory systems. Usually, all the necessary OMP libraries are already included in the GNU compiler on Linux. Nevertheless, it might be necessary to install additional packages like *gcc-multilib* or *libomp-dev*. The easiest way to check whether OMP is available on your machine is to include *omp.h* into your project.

The basic workflow with OMP is by using predefined directives of the form **#pragma omp** to parallelize blocks of code with threads. The number of threads will be derived from the number of logical processors, if not explicitly stated. In fact, we can request all relevant information about the resources at hand by calling predefined functions such as: *omp_get_num_procs* (number of logical processors), *omp_get_num_threads* (number of available threads), *omp_set_num_threads* (defines the number of threads to be executed) and so on.

OMP is frequently used to parallelize for-loops with the directive **#pragma omp parallel for**. As a result, the iterations to be repeated will be equally divided between threads and executed in parallel. The original iteration order is not preserved because of various optimizations by the operating system's process scheduler, who organizes the execution. If the order is important for us, we can apply **#pragma omp ordered** to keep iterations in line. For any directive to work, we need to pass *-fopenmp* flag to the compiler.

OMP supports thread synchronization in three different ways: we can declare specific code blocks as a critical region, we can flag one concrete operation as atomic or we can use the barrier. Indeed, the barrier is implicitly put at the end of every parallelized for-loop. When encountering a barrier, the program will pause and wait until all threads finish their workload up to this point before proceeding with the code execution. This behavior can be changed with **nowait** directive.

2.2 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is an application programming interface and a parallel computing platform designed by Nvidia. CUDA provides direct access to

the GPU's virtual set of instructions and elements, which allows utilization in general-purpose program development. It uses a *nvcc* compiler and can handle C/C++ for the most part, some features might not be supported though. At HTW Berlin, we have ready to use *deepgreen* machines with dedicated *Nvidia Tesla M40* GPUs.

During the programming process, we can write our own kernel functions, which can later be executed in one of the various GPU threads. Since the memory is distributed, we need to copy the *in computation involved* data from CPU *host* to GPU *device* at the beginning and back after the results are calculated. CUDA has a hierarchical thread-block-grid structure, where each level can access its own memory and the memory from the level above. For instance, each thread can access its own local memory and the shared memory of the parent block. *Nvidia Tesla M40* mentioned earlier, allows at most 1024 threads per block.

The workflow with CUDA consists of several basic steps. First, we need to change the source file extension to **.cu** because of the special compiler. Second, we need to include *cuda_runtime* and *device_launch_parameters* headers into the file. Third, we need to take care of memory allocation and data distribution. Particularly the later step introduces supplemental programming overhead.

CUDA is also frequently used to parallelize for-loops. This can be done by requesting the unique identifier of each thread, which then replaces the increment loop-variable. The total number of threads and blocks is defined during the kernel call. If we pass the limit-condition-value there, we can check it from inside the kernel. Equipped with this construct, we can replace a for-loop by thread kernels, who simultaneously process one iteration code. However, CUDA is bad in creating and following logical flows and when it comes to synchronization, there is just one barrier like element. Thus often a complete code redesign is necessary if we want to migrate our existing code to CUDA.

2.3 Dijkstra's Algorithm

Dijkstra's algorithm (or simple Dijkstra) finds a solution for the SSSP based on a graph. A graph typically consists of a set of vertices V, together with a set of edges E and can be directed, weighted, or both. Graphs are usually represented by either an adjacency list or an adjacency matrix. An adjacency list can be abstractly imagined as a list of lists, where each first level list position corresponds to a specific vertex. Under each position, another list of all edges of this vertex is to be found. Thus, we have a maximal space consumption of $\Theta(V + E)$.

An adjacency matrix looks like an ordinary multiplication table, with source vertices on the vertical side and target vertices on the horizontal line. It consumes $\Theta(|V|^2)$ storage. When the vertical and horizontal lines cross and the filed holds a non zero value, it indicates a connection between the involved vertices. In a weighted graph, the value

holds an exact connection weight. For dense graphs, both data structures consume the same amount of space. However, real-world graphs such as road networks are not dense. Furthermore, even if the graph is dense, an adjacency list can yield the set of neighbors quicker than a matrix.

The algorithm was originally conceived by a dutch computer scientist Edsger Wybe Dijkstra in the late 1950s, it is one of the most prominent cornerstones in graph theory. Due to this fact, there are numerous improvements and variations of Dijkstra with different time complexities and structures. The original version, for example, runs in $\Theta((V+E)*\log V)$, while an implementation with arrays requires $\Theta(V^2+E)$. In general, we can only process edges with positive weights. Also, Dijkstra is a greedy algorithm, so that the part-solution at hand is always optimal.

3 Conception

This chapter describes the planning phase of the project. We look at the program structure and data composition. We shall examine different Dijkstra versions, how they are parallelized and synchronized. Before we can dive into that, though, we begin with common program elements, shared between all implementations.

3.1 Common elements

All Dijkstra versions will take a graph and a source vertex as an input. After the calculation, they will return the found vertex connections and the corresponding weights. Since the graph directly affects the program run time, we intend to develop a stand-alone graph reader function. Along with other auxiliary functions, it will be put in a separate file *data.cpp* for reuse among all versions according to Table 1.

Function	**Description**
printAdjacencyList	prints adjacency list
getShortestPathToX	unfolds path from adjacency list
readGR	reads .gr file from disk
split	splits a string line on whitespace

Table 1: Auxiliary functions in data.cpp

The Auxiliary functions will be linked to the main program through the header file *data.h*. Here we will also declare our custom data structures as summarized in Snip. 1. The graph will be represented by an adjacency list in the form of a map with vertex as a key and a vector of neighbors as value. As a result, we can access the same involved

```cpp
typedef int vertex_t; // vertex=node
typedef double weight_t;

struct neighbor {
    vertex_t target;
    weight_t weight;

    neighbor(vertex_t arg_target, weight_t arg_weight)
            : target(arg_target), weight(arg_weight) {}
};

const weight_t max_weight = std::numeric_limits<double>::infinity();
typedef std::map<int, std::vector<neighbor>> adjacency_list_t;
```

Snippet 1: Custom Data Structures in data.h

vertex connections during the read, even if they appear in different parts of the file. The neighbor is a custom structure to hold the target vertex and its weight in one single object. Moreover, we will define aliases for target vertex from integer type and for the weight from double type to achieve better logical recognition.

The code around Dijkstra will be organized in the main function, which will be responsible for variable initialization, argument parsing, time tracking, and path printing at the calculation end. Arguments should serve to dynamically define the graph file at execution time, as well as to set the shortest path start and end vertexes. An overall overview of the relationships between files is demonstrated in Fig. 1.

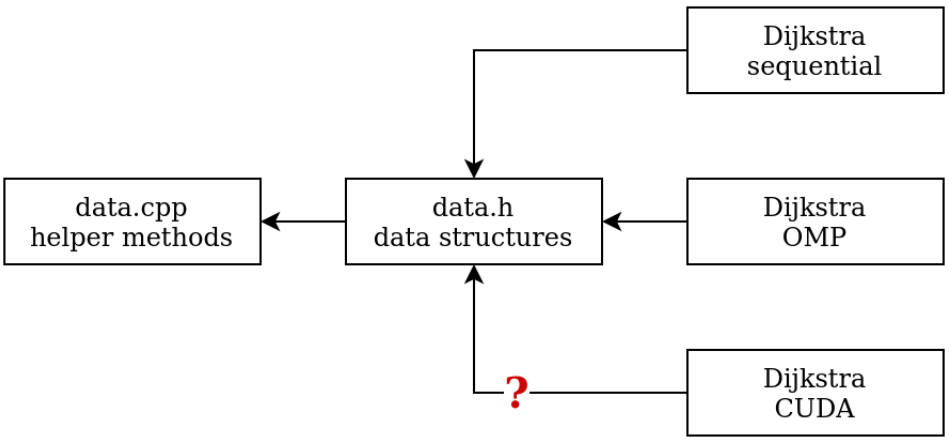


Figure 1: Project Structure

3.2 Dijkstra, Sequential & OMP

The sequential Dijkstra and its OMP counterpart are going to be planed together since they will share the same fundament and can utilize the same C++ elements. The former version is going to rely on a queue to process vertexes and on a neighbors for loop to inspect the outgoing connections according to the pseudocode in Snip. 2. The essential modification for OMP consists of paralyzing the neighbors for loop with **#pragma omp parallel for**. Consequentially we need to take care of synchronization when accessing output vectors for distance and previous vertex. There are two options to try out — we can perform relaxation in a critical region or, we can set a lock.

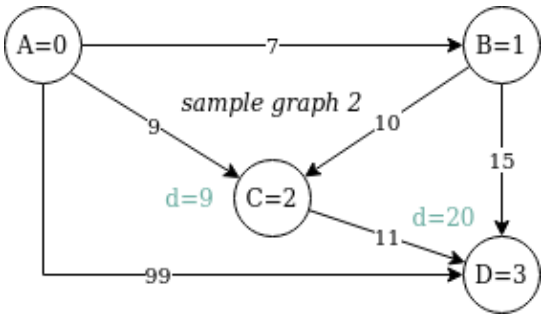
```
1  function Dijkstra(G, source):
2    create vertex queue q
3
4    for each v in V: // initially we consider V to be unreachable
5      dist[v] = inf
6      prev[v] = undefined
7      q.add(v)
8    dist[source] = 0
9
10   while Q is not empty: // main vertex loop
11     v = take next vertex from Q // in first run-through, v is source vertex
12
13     for each neighbor u of v: // inspect every neighbor u of v
14       alt = dist[v] + length(v, u)
15
16       if alt < dist[u]: // shorter path from v to u has been found
17         dist[u] = alt // relaxation
18         prev[u] = v
19         q.add(u)
20
21   return dist[], prev[]
```

Snippet 2: Pseudocode of Dijkstra

For the queue, we use STL's self-balancing binary tree — set. Thus we can insert or remove vertexes with the same logarithmic time complexity. We keep output data for neighbors/distances in separate vectors and initialized them at the beginning with infinities. Then, the weight of the source vertex is set to be zero as it is our starting point, first to be loaded into the queue. Other vertexes are going to be loaded at most once because by the next run they will not satisfy the relaxation condition of achieving a shorter distance as present in the distance vector. The inner loop executes at most once for every vertex neighbor with the chance of reordering the queue. As a result we have a total time complexity of $\Theta(E * \log V)$,

3.3 Dijkstra CUDA

Dijkstra version for CUDA requires much more adjustments than OMP. First, due to CUDA's peculiarities, it is difficult to say, whether the auxiliary methods from *data.cpp* could be reused, that's why there is a question mark in Fig. 1. Second, *Standard Template Library* (STL) of C++ is not available so that we need to redesign our graph representation without using a map for the adjacency list.



	0	1	2	3
0	-1	7	9	99
1	-1	-1	10	15
2	-1	-1	-1	11
3	-1	-1	-1	-1

The representation is changed into a sparse adjacency matrix in the form of a single integer vector. To illustrate this, let's consider *sample graph 2* with four vertexes and six edges. In the formation of an adjacency matrix, it would have a 4×4 size with connections going from lines to columns. When converted into a sparse form, every connection gets two slots in the array, one for the vertex and another for the weight, resembling the neighbor construct from Snip. 1 as pictured in Fig. 2. The colored line strips on top signify vertexes, the middle green line shows the imaginary flattened index and the bottom line target-weight pairs. As a result. the graph can be placed into one single vector, but we need additional effort to intermediately calculate the correct index before accessing the data.

0						1						2					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	7	2	9	3	99	2	10	3	15	-1	-1	3	11	-1	-1	-1	-1

Figure 2: Sparse matrix representation of *sample graph 2*

In a sequential Dijkstra, we used a queue, which is also a part of STL, to keep track of processed vertexes. Now that it is no longer available, we need to think of another solution. This will be an additional vector to mark vertexes as visited. The pseudocode of modified Dijkstra is illustrated in Snip. 3, it primary targets dense graphs. Since it is not clear, what will work in CUDA, we will proceed by gradually implementing the code in a plain *dijkstrac.cpp* first and only then move program blocks to CUDA, preserving functionality.

As with the sequential version, we initialize the calculation relevant vectors for neighbors, distances, and visited statuses; the source vertex distance is set to zero afterward. Again, we have two loops structure with the outer loop termination being hooked upon the condition that all vertexes have been visited. Between all the available, not visited

```
function Dijkstra(G, source):
  for each v in V: // initially we consider V to be unreachable
    dist[v] = inf // and not visited yet
    prev[v] = undefined
    visited[v] = false
  dist[source] = 0

  while there is at least one vertex with dist < inf:
    v = vertex with min dist and not visited
    if dist[v] = inf:
      return
    visited[v] = true

    for each neighbor u of v:
      alt = dist[v] + length(v, u)

      if alt < dist[u]: // shorter path to u has been found
        dist[u] = alt // relaxation
        prev[u] = v

  return dist[], prev[]
```

Snippet 3: Pseudocode of Dijkstra for CUDA

vertexes, we always choose the one with the least distance and proceed to the neighbor examination. Because our graph is represented as a sparse matrix, we must skip empty vector slots. Moreover, to eliminate infinite loops, we must skip iterations from one vertex to itself ($u = v$) as well. Therefore, with the relaxation remaining unchanged, the total time complexity equals to $\Theta(V^2)$.

The main design challenge so far — how to meaningfully split vertex and neighbor loops. Also, it would be performance-friendly, if we could call the kernel just once, passing the required arguments and let the threads do their calculation. To achieve this, we need to differentiate between the first calculation part where we identify the next vertex for examination and the second part where we go through all its neighbors. The second part depends on the finish of the first part. If we declare the vertex to be examined as a shared variable, all threads can see it, which is crucial for the path calculation. However, without a critical section or a lock as in OMP, we need to limit the code execution of the first part to only one thread. Then, we can ensure that all threads wait until the vertex is identified by setting a barrier **__syncthreads** before entering the second part. There, each thread processes exactly one neighbor, according to its ID, which offers a security mechanism while writing in the output vectors during the relaxation.

4 Implementation

This chapter outlines crucial implementation cornerstones, aroused difficulties we could not foresee in the planning phase and the way through.

4.1 Dijkstra, Sequential & OMP

The program's main function accepts three arguments at execution time: graph file, source, and destination vertexes. If nothing is provided, we fall back on the predefined set up with a sample graph. A notable fact during the adjacency list implementation was — the default data structure in C++ seems to be not an array, but a vector. On the internet, people tend to suggest using a vector, rather than starting a lengthy explanation about how to allocate, access, and free array memory.

The time measurement is split into initialization and calculation parts, as it can take a considerable time to read the graph from disk. We use *clock_t* for timing purposes: one is triggered at the beginning, and another is wrapped around Dijkstra execution. It is arithmetic, real type that represents the processor time used by a process. Accordingly, it must be pretty formatted by *CLOCKS_PER_SEC* for the print output.

Once, during the implementation of Dijkstra OMP, there was an unexplained compiler complaint about a missing } (curly closing bracket). All the openings and closings were thoroughly double-checked without any results. So in the next debug step, all brackets were put on a new line in a C++ fashion, which successfully compiled. It turns out — OMP needs the brackets to be formatted in a C++ style indeed.

Initially, the relaxation synchronization was done by **#pragma omp critical**, with Valgrind being the test tool against race conditions. However, for accurate work, Valgrind needs the *libgomp.so* to be rebuilt with the **–disable-linux-futex** option and to be then included in the program build process. This was an experience, one doesn't often make as a Java practitioner. Later, **#pragma omp critical** was replaced by **omp_lock_t** for performance tests and resulted in many reported false positives from Valgrind — this was a setback, we didn't know how to handle for a while. Finally, following the professor's advice, the reason was found in different sized lock structures of OMP in the rebuild *libgomp.so* and the standard one. Part of the solution was to point the compiler to the altered OMP header during the build. Another part was to switch from Valgrind to thread sanitizers for race condition detection with **-fsanitize=thread**. Thus we also switched from Valgrind's dynamic instrumentation to static instrumentation of thread sanitizer, which can be regarded as a disadvantage.

4.2 Dijkstra CUDA

Even though CUDA provides a concept of code and header files similar to C++, it mainly serves the purpose of calling CUDA functions from C++. Thereby it's not possible, to reuse already developed utilities. Also, the lack of STL elements, such as streams and strings, makes our graph reader inapt anyway. After a quick internet research **fscanf** was found, which promised to read the data from the stream and store it, according to the provided format, into the desired variables. I.e. we can read the graph file line by line and retrieve only relevant information like arc direction and weight. However, after the implementation, we observed that **fscanf** reads not by line, but by submitted format. This led to a gibberish output and the necessity to consider all possible line structures. So, another research attempt was made to find a more reliable alternative. Thus the C functions **getline** & **sscanf** were discovered and the new reader emerged.

The reader was organized within a while block, except for the first headline read where the graph size is stated. The read is done by **fscanf**, after the heads have been manually checked on stricture compliance. As a termination condition, we used **getline**s return status — when it's equal to minus one, we quit the loop. Also, now that **getline** really reads file line by line, we can store them as a character sequence and process arcs only. We use the return status of **sscanf** to ensure the line structure correctness there. In the loop, the data is first pipelined into the old map based adjacency list. Besides, here we determine the maximal number of outgoing edges per vertex. We then use this information in the second step, to fill the sparse matrix, as an offset. More specifically, we loop through the sorted adjacency list and copy the connections into our sparse vector. This way, we don't need an additional container to store fluctuating offsets. It is not the most efficient technique, but it is good enough for our use case since the main interest lies in the comparison of Dijkstra performance.

CUDA code can be divided into three major blocks: main, Dijkstra, and kernel. In the main function, we initialize host data structures and call Dijkstra. Moreover, here we enjoy another advantage of using vectors instead of arrays by declaring the size and default fill values in one single call to **resize**. Dijkstra handles memory allocation on the device, copies the data there, and calls the kernel. The kernel function is executed on the GPU and performs the actual shortest path calculation. It is called with one thread per neighbor within a single block, using the maximal number of outgoing vertexes we previously discovered during the graph read. It is worth noting — CUDA cannot handle while loops because it is not obvious when they are going to terminate. Afterward, we transfer the data all the way back from the device to the host, where the computation results are printed on the console.

One of the spontaneous experiments, not mentioned in the planning phase, was about the trial to execute the program with an adjacency matrix. Although in the flattened form, it still requires $\Theta(|V|^2)$ storage space. The expected result was that we run out of

memory, in reality, a *Segmentation fault (core dumped)* was thrown. Let's look at the example numbers to understand involved space capacities: a graph with 100 thousand nodes & 10 outgoing edges on average would consume one million slots, if it would be an adjacency list and ten billion slots if it would be an adjacency matrix. No doubt, that enormous difference gap could be overwhelming.

The overall experience of moving program blocks from plain C++ to CUDA was very tedious. There were little things that didn't work for some mysterious reasons all the time, a couple of examples discussed above are only the tip of the iceberg. This resulted in injections of many additional helper prints located at different points. To turn them on or off, an extra Boolean argument was added to the program.

5 Evaluation

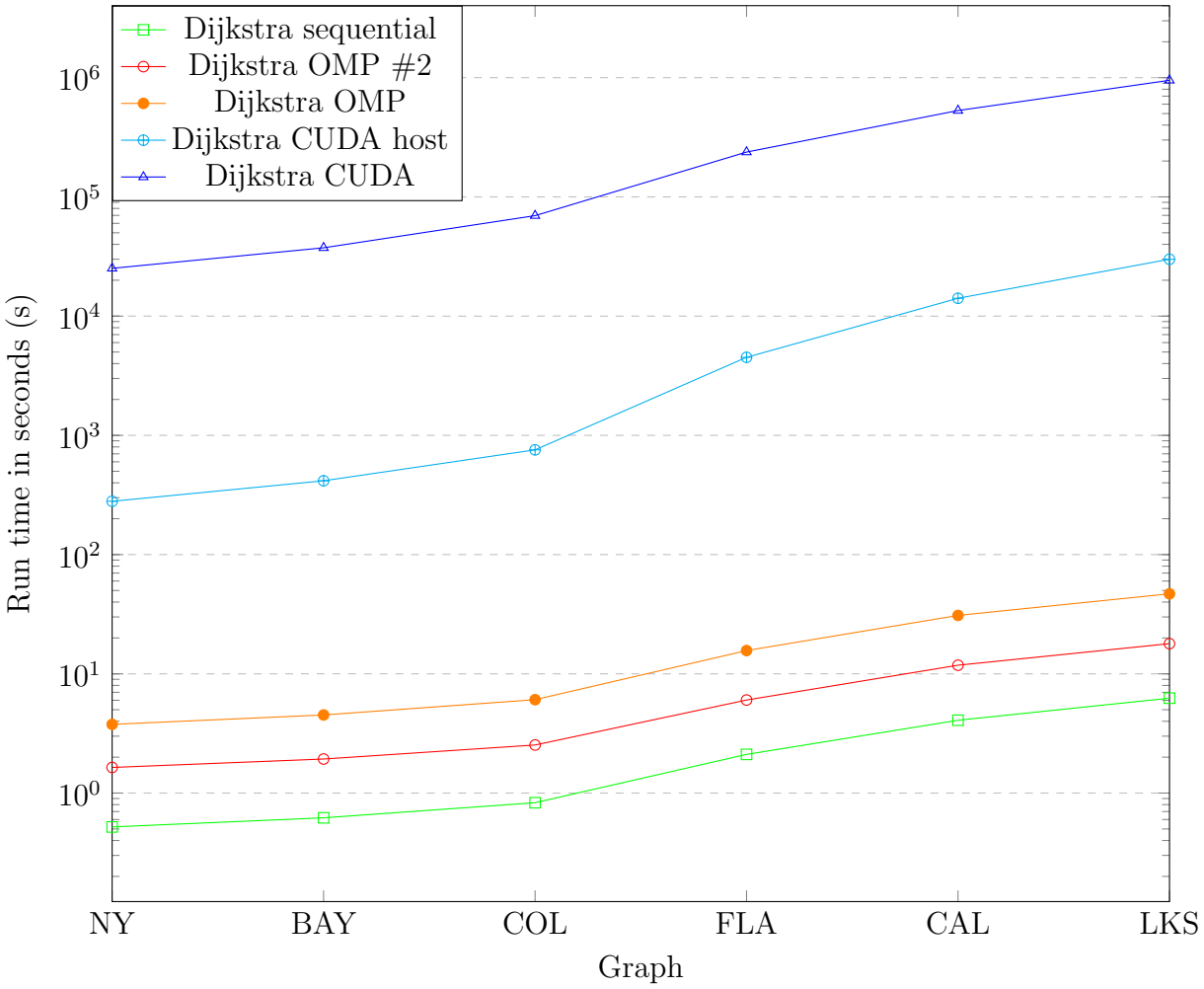
This chapter describes run time tests of implemented Dijkstra versions. They are based on Dimacs graphs found under http://users.diag.uniroma1.it/challenge9/download.shtml, which represent different road networks in the USA and are characterized in Table 2 ordered by file size in megabytes. The graphs are organized in a text file with the headline containing the graph size with the number of nodes and edges. A headline for *sample graph 2*, we used as an example in Section 3.3 looks like: "p sp 6 9". Each following line starts either by *c*, which indicates a comment or by *a*, which stands for an arc. Every arc contains the information about where it starts, where it ends, and what the corresponding weight is, for instance — "a 3 4 11".

Name	Region	# nodes	# arcs	Size MB	Target
NY	New York City	264,346	733,846	14	25,907
BAY	San Francisco Bay Area	321,270	800,172	16	321,270
COL	Colorado	435,666	1,057,066	21	435,666
FLA	Florida	1,070,376	2,712,798	54	1,070,376
CAL	California & Nevada	1,890,815	4,657,742	96	1,890,810
LKS	Great Lakes	2,758,119	6,885,658	144	2,757,872

Table 2: Summary of test graphs

To ensure the path can bridge enough vertexes during the calculation, we always used the vertex #1 as the source, while the vertex depicted in the last column was the target. The maximal number of outgoing edges per vertex is always equal to or lower than nine. The test system specifications are as follows: Intel Core i5-3320M CPU, 16GB RAM, hard disk drive, and Ubuntu Bionic OS. The processor has two, up to 3.30GHz cores with two threads per each core. Nvidia specs were previously reported in Section 2.2.

The diagram below illustrates the results of five tests, which were run in total. The initialization time variated between 1–40 seconds and is not included in the diagram. On the x-axis, we have the graph ticks, the y-axis reflects the corresponding run time in a logarithmic scale (because of wide-ranging values). The outcome of the sequential Dijkstra is shown in green — the completion of the NY graph took half a second, and great lakes were finished after about six seconds. The time consumption rises steadily, except for the frog leap between Colorado and Florida. This was expectable since the later is two and a half times greater in size and number of arcs.



Dijkstra OMP test with full CPU utilization, using 4/4 threads, appears in gold. Surprisingly it consumes more time than the sequential version with the run time varying between 4s for NY and 48s for the great lakes, which is eight times longer. The program was then run with 2/4 threads again — see red line. Still more astonishing, it snatches between the first two results: it terminates about three times faster than with four threads and three times slower than the sequential Dijkstra. Evidently, the ex-

tra threads do not lead to better performance, but rather slow the calculation down. This was confirmed by one more run with three threads. Regarding the relaxation synchronization — marking operation as atomic did not prove to be faster than a critical section.

The CUDA's algorithm was benchmarked on the device and host sides to create a relation to the sequential Dijkstra. Completion of the NY graph on the host side took about six minutes, and LKS were finished after 500 minutes — see cyan line. This was to be expected since we have a quadratic time complexity, which is clearly greater when compared with the sequential version. The result from the device side is illustrated in blue: NY seven hours and Florida 66 hours. The exact execution time for CAL & LKS graphs could not be measured to the end due to the project deadline and was instead extrapolated from the existing numbers so far. In any case — the numbers totally deviate from the expected outcome because we were counting on threads concurrently checking all the neighbors at once, thereby decreasing the total run time to something like $\Theta(V * 8)$. With maximal eight neighbors per vertex, that would have been a notable speed up.

6 Conclusion

In this chapter, we are going to summarize key project points and give an all-round retrospective on what had been accomplished.

6.1 Results

Most significantly — the Dijkstra's algorithm was implemented sequentially, as well as using the parallel interfaces. The performance comparison between these versions was carried out too. However, none of the developed programs yielded the initially expected results. OMP interface, being closely integrated with C++, did not bring the anticipated speed up with its **#pragma omp parallel for**, neither did the switch from critical section to atomic operation. Moreover, a greater value of **OMP_NUM_THREADS** negatively affected the overall program performance.

CUDA development was certainly a great opportunity to get hands on a specific problem and learn by trying to solve it, despite the results. If we would try to sell them in a free market, it's hard to imagine someone paying for them. Nevertheless, the implemented algorithm worked in principle and was slowly but steadily processing road graphs. We saw, what an influence the underlying data structures have on the program by experimenting with adjacency lists and matrices. We also learned about the device's shared

memory, which can be dynamically or statically allocated yet always stretches as a consecutive line of memory addresses, so that it's necessary to manually place vectors there and take care of access indices too. This wasn't explicitly described in the document but was tried out at some point. In the end, it was sufficient to have one single shared integer.

While all used C language elements were supported, STL of C++ wasn't. Thus additional work is required to shift C++ code to CUDA, which is a pity but might well be changed in the future, according to Nvidia. Equipped with the synchronization barrier only, we often need to rethink or redesign the original program. Still, sometimes we cannot implement exactly the same control flow as we would otherwise have on the CPU side. As a consequence, CUDA seems suitable for a monotonic process of big amounts of data with the supervision of CPU.

6.2 Retrospective

I think the solo project execution was a good choice because my friends were mostly skipping this remote corona semester anyway. To work with someone I don't know, would be too much communication overhead. Writing emails back and forth or zooming/skyping is just not the same as sitting together in one room and talking to each other.

The choice of Dijkstra's algorithm might not be the optimal one for a one-person team. During the 162,5 project hours, I could learn C/C++ on the fly reasonably fast, but at times it felt like the pointer reference concept or the lack of an intelligent IDE were burning too much time. Understanding the theoretical background was also a time-costly activity.

I wish I'd start by running someone else's SSSP programs with Dimacs graphs to establish a global reference point from the beginning on, rather than developing the wheel again. Particularly a comparison of vector-based vs. array-based graph structures would be helpful for performance understanding. I tried it in the evaluation stage, but they wouldn't run out of the box, and there was not enough time left to make any adjustments. So, I guess, I rushed to the coding phase too quickly, instead of diving deeper into planning and researching first.

To come back to the initial project objective — I think SSSP can be effectively parallelized, provided the graph structure is based on an array, and you have expert knowledge about the interface you are using with all its *twerks, twirls and twists*.