Kacper Walentynowicz

Parallel route-finding algorithms for road networks

Computer Science Tripos – Part II

Trinity College

2021

UNIVERSITY OF CAMBRIDGE

Declaration

I, Kacper Walentynowicz of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed

Kacper Walentynowicz

Date

13.05.2021

Proforma

Candidate number:	**2339B**
Project Title:	**Parallel route-finding algorithms for road networks**
Examination:	**Computer Science Tripos – Part II, 2021**
Word Count:	**10426**[1]
Code line count:	**6318**[2]
Project Originator:	Dr Jagdish Modi
Project Supervisors:	Dr David Greaves, Dr Jagdish Modi

Original Aims of the Project

The original aim of the project was to study several existing approaches to extending known sequential shortest path algorithms for suitability for efficient parallel computation, and to provide their implementations. Because of difficulties in evaluating parallel algorithms, a goal of the project was also to build a simulator which would enable easy implementation and fair comparison of the algorithms.

Work Completed

The original aims of the project have been completed. I have successfully implemented a tool which allows simulation of algorithms on various parallel architectures. Then, I have met the other goals of the project by implementing and evaluating several parallel adaptations of known shortest path algorithms: Dijkstra's algorithm, Matrix multiplication algorithm and SPFA algorithm. I have then compared the parallel approaches taking into account both theoretical merits and disadvantages, as well as empirical evaluation on real-world road networks.

Special Difficulties

None.

[1] counted with Overleaf's builtin feature using TeXCount

[2] counted with `git ls-files | xargs wc -l`. There are 5702 lines in the source directory and 1198 lines in the test directory. 582 lines of source code written by external authors needed to be subtracted.

Contents

Chapter 1

Introduction

How much algorithm redesign is needed for parallel architectures? It is certainly not an easy question to answer. For sequential machines, various No Free Lunch theorems [13] establish that there is no algorithm perfect for all applications. This phenomenon is particularly visible in parallel systems where communication or synchronization costs may outweigh advantages arising from parallelization, especially if the problem instance is not large. Although there exist attempts at writing software capable of performing efficient parallelizations (such as MIT's pMapper [12]), designing efficient parallel algorithms remains an important challenge. In my project, I have chosen several known sequential algorithms for the shortest path problem and considered the tradeoffs associated with their parallel adaptations.

1.1 Importance of parallel algorithms

During several previous decades of computing, hardware engineers have been able to deliver better processors as quickly as Gordon Moore had anticipated in the 1970s. Recently, it seems like the boundaries can no longer be pushed as the Moore's Law has slowed down[6], and programmers will need to resort to alternative ways of achieving speedup. The natural solution is using multiple workers; cores on a single machine or simply several machines communicating together. The most important benefits are energy efficiency and scalability.

The relationship between clock frequency and power used is non-linear, and therefore to achieve the same performance a single machine needs more energy than a parallel setup. Additionally, machines are suffering from von Neumann bottleneck. The processor and memory are separate, with data moving frequently between them. Latency increasing is an unavoidable consequence of physical distance between destinations, resistance of wires, and other key factors.

Importance of scalability in distributed systems also cannot be underestimated. The cost of adding new machines to an already existing system is very cheap compared to improving processor designs. However, parallel systems generate numerous other difficulties which I'm discussing in section 1.3.

1.2 Shortest path problem

In my project I am studying shortest paths as an important example of fundamental computer science algorithms. Shortest paths algorithms play an important role in graph theory, also as a building block in more complicated algorithms (such as Minimum Cost Maximum Flow). Additionally, they are prevalent in other domains such as internet routing, route-planning (for vehicles), or planning in robotics.

Formally, let $G(V, E)$ be a graph with vertex set V and edge set E. For our purposes, each edge i $(1 \leq i \leq |E|)$ is of the form (a_i, b_i, w_i) and allows moving from vertex a_i to b_i which takes time w_i.

A path between two nodes (S, T) is therefore a sequence of edges e_1, e_2, ..., e_k such that the following hold:

- $a_{e_1} = S$
- $\underset{2\leq i\leq k}{\forall} a_{e_i} = b_{e_{i-1}}$
- $b_{e_k} = T$

Naturally, the shortest path p_{opt} minimizes $\sum\limits_{1\leq i\leq k} w_{e_i}$ among all paths p which satisfy the constraints above.

There exist two main classes of traditional sequential shortest path algorithms: Single-Source Shortest Paths and All-Pairs Shortest Paths. SSSP algorithms calculate for a given node the shortest distances to all the other nodes, while APSP algorithms simply compute the distances between all possible pairs of sources and destinations simultaneously.

1.2.1 Single-source shortest paths

The most famous shortest path algorithm is arguably the Dijkstra's algorithm which is dated to late 1950s, and described in section 2.3.2, in the Preparation chapter. Another known shortest path algorithm is named after Bellman and Ford. It has worse theoretical complexity than Dijkstra's, but works even if the edge weights can be negative. Somewhere in the middle between these two lies a simple, yet relatively unknown Shortest Path Faster Algorithm (SPFA) described in detail in section 2.3.3. The algorithm appears to not have been extensively studied by theoretical computer scientists, but has certain features which should allow an efficient parallel adaptation. Testing whether parallel SPFA can offer a simple and efficient solution for shortest path problems was one of the reasons which inspired me to delve into this project.

1.2.2 All-pairs shortest paths

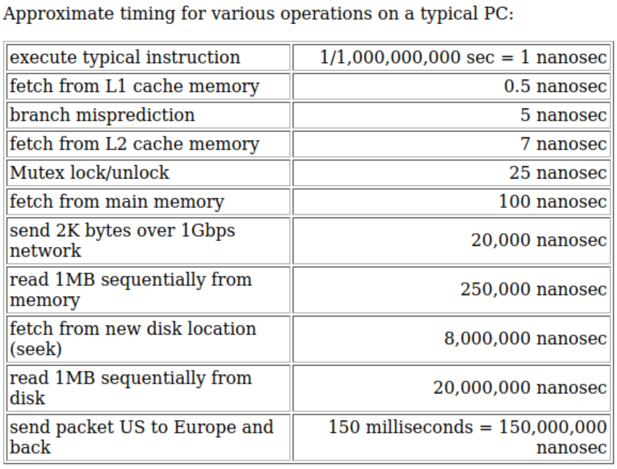
Another approach which proves efficient especially for dense graphs is to calculate shortest paths in a way which resembles matrix multiplication. Matrix multiplication exhibits a very high degree of parallelism, and therefore this simple method may have good parallel adaptations. Alternatively, one can also consider running one of the above SSSP algorithms $|V|$ times, once for each possible source vertex.

1.3 Challenges in parallelization

Parallelization of an algorithm is not an easy task. The two challenging issues are explained in the subsections. These are not the only ones, and there exist numerous others, for example the name alias problem.[1] Although they are of primary importance in the topic of autoparallelization, they are less relevant to my project.

1.3.1 Importance of system architecture

Suppose we are doing a computationally expensive task on a system with several workers who carry out tasks. In a perfect world, every worker would be able to perform their own work independently, and only report the result once they finish. Unfortunately, life is not perfect like that, and sometimes the workers need to collaborate - share data, report intermediate results, access the same resources, and so on. The costs of these operations may vary **orders** of magnitude, depending on what are the building blocks of our system. The following table shows the comparison among various possible operation costs[2].



Approximate timing for various operations on a typical PC:

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

When talking about classical sequential algorithms these costs are often neglected. In parallel systems communication of workers and their utilization are crucial for efficiency, and therefore need to be carefully considered.

1.3.2 Loop–carried dependency

One of the greatest challenges in automatic parallelization is loop-carried dependency, studied in detail for example in the paper *'Dependency Analysis and Loop Transformation Characteristics of Auto-Parallelizers'*[11]. Some algorithms are notoriously difficult to parallelize due to their sequential nature of operation – later iterations of a loop depend on earlier ones. A parallel adaptation of a highly efficient sequential algorithm may be

[1] It is not easy to determine which computations can be safely run in parallel. They may reference the same array indices which results in race conditions. This touches the undecidability of arithmetics problem.

[2] Author: Peter Norvig, director of research at Google. Table taken from the article: '*Teach yourself programming in 10 years*'

truly disappointing if workers are idle majority of the time. On the other hand, simple algorithms may exhibit natural parallelism. Old algorithms which predate computing era are especially promising, as they were not designed with execution on serial machines in mind. The difference between the shortest path algorithms studied is striking – Dijkstra's algorithm is an example algorithm of the first type, while SPFA – of the second one. Is maintaining sequential consistency at all costs a good idea, or is it better to relax the constraints? Which way is faster and saves more energy? These are the questions which I attempt to answer.

Chapter 2

Preparation

2.1 Starting point

Prior to the project, I did not have much experience with parallel programming. Some principles and methods have been outlined in *Concurrent and Distributed Systems* part IB course.

Starting point of the project was nearly from level zero. The only code which I had implemented were sequential versions of Dijkstra's and SPFA algorithms to serve as a proof-of-concept that SPFA can compete with Dijkstra's algorithm in terms of performance even in the standard setup of sequential machines[1].

2.2 Requirements analysis

As described in the introduction, parallel algorithms are an interesting, but challenging topic. Because performance of algorithms is heavily influenced by system architecture, my project needs to consider these. Following initial analysis of the problem, I have decided that a key component of the project would be a simulator providing an evaluation environment for the algorithms.

The tool needed the following features:

1. Allow the abstract description of a parallel algorithm to be easily translated into a working code.

2. Provide functionalities for cores - entities which are going to perform work of our algorithms. These include basic primitives such as ALU[2] operations, accessing local memory of the cores, as well as simple data structures for those: local arrays, queues, and sets.

3. Provide means of communicating for the cores - either by passing messages (broadcast or core to core), or accessing shared data structures in memory.

[1] See the section 2.3.3 for details

[2] Arithmetic Logic Unit

4. Allow specifying the topology of interconnections to reflect that in real-world machines there is limited number of direct connection between workers.

5. All of the operations should be implicitly tracked by the evaluation environment. The programmer should also be able to specify what the costs of various operations in the system are in order to test the algorithms in different circumstances.

6. Running the simulations using multiple threads of a physical machine, so that the computations do not run unreasonably long.

7. Allowing to run the simulations in single-threaded debug mode. There are lots of surprising concurrency bugs which can arise from incorrect implementations and are not easy to find. Therefore, a debug mode which allows deterministic execution could prove very useful.

2.3 Algorithms

In my project I have studied parallelizations of three different standard algorithms for shortest paths, and evaluated their relative tradeoffs. In this section I am briefly describing the sequential versions and the ideas associated with their efficient parallelizations. The exact division between *Preparation* and *Implementation* chapters is somewhat blurred, as understanding the algorithm, its exact requirements and limitations is the key part. Afterwards, translating the algorithm into the code running on the simulation environment should be rather smooth.

In the descriptions of the algorithms I am assuming the standard conventions: the graph $G(V, E)$ has N vertices labeled from 1 to N and M edges of non-negative weights. If a path between the source and destination does not exist, its length is ∞. In parallel algorithms, workers (called interchangeably 'cores') are labeled from 1 to C.

2.3.1 Matrix multiplication

The first algorithm implemented was an algorithm based on matrix multiplication, let's call it MatMul for short. It is an All-Pairs Shortest Paths algorithm which calculates the answer in the array $dist[N][N]$. Its operation consists of $log(N)$ steps of dynamic programming.

In the initialization phase, we calculate the shortest distance assuming that we are allowed to traverse no more than 1 edge - the adjacency matrix.

Now, let's assume we want to know what the shortest paths are if we are allowed to traverse no more than $2K$ edges, for some K. What does a path from a to b that uses no more than $2K$ edges consist of? A path from a to some intermediary node c of no more than K edges, followed by a path from c to b, also of no more than K edges. We don't know what c is, but we can simply consider all the options!

Therefore, given $dist_K[N][N]$ we can easily calculate $dist_{2K}[N][N]$. The following pseudocode solves our shortest path problem.

Algorithm 1 MatMulDynamicProgramming

```
procedure MatMulDP(G)
    dist ← G.adj_matrix()
    newdist ← dist
    for $step \leftarrow 1$ to $log(N)$ do
        for $a \leftarrow 1$ to $N$ do
            for $b \leftarrow 1$ to $N$ do
                newdist[a][b] ← dist[a][b]
                for $c \leftarrow 1$ to $N$ do
                    newdist[a][b] ← min(newdist[a][b], dist[a][c] + dist[c][b])
        dist ← newdist
    return $dist$
```

One can notice that three innermost loops resemble matrix multiplications with a switch of operators - sum over all the elements is substituted with min, and we are adding two terms rather than multiplying. Talking formally, it is just matrix multiplication over different algebraic fields.

Algorithm 2 MatMul

```
procedure MatMul(G)
    dist ← G.adj_matrix()
    for $step \leftarrow 1$ to $log(N)$ do
        dist ← multiply_matrix_by_itself(dist, (+,min))
    return dist
```

It is very clear what needs to be done to obtain an efficient parallel algorithm - write good matrix multiplication! For that, we can use Cannon's algorithm [3], which ensures that there is little unnecessary movement of data between iterations. The operation of Cannon's algorithm is described in the *Implementation* section of MatMul, which would be almost empty if I included a description of Cannon's algorithm here.

2.3.2 Dijkstra's algorithm

Dijkstra's algorithm is one of the most well known computer science algorithms taught in every course on standard algorithms, and therefore just recalling its pseudocode (Algorithm 3) will suffice. The operation of updating the distances of nodes is usually called *relaxation.*

There is a notable difficulty in parallelizing this algorithm - in line 7 the nodes are being taken one after another in different iterations. Data dependencies arise, because changes in *dist* may affect which nodes are extracted from the queue in the subsequent iterations. Nonetheless, there have been attempts at efficient parallelizations.

Q is a priority queue, typically implemented with a Fibonacci heap to guarantee $O(|E| + |V|log|V|)$ runtime. A simple approach for a parallel speedup would be to split the slow computation of selecting the vertex with minimal distance.

Algorithm 3 Dijkstra's algorithm

```
 1: procedure DIJKSTRA(G, SOURCE)
 2:     for $node \leftarrow 1$ to $N$ do
 3:         $dist[node] \leftarrow \infty$
 4:     $dist[source] \leftarrow 0$
 5:     $Q \leftarrow G.nodes()$
 6:     while not Q.empty() do
 7:         $u \leftarrow$ node in $Q$ with minimal $dist[]$
 8:         $Q \leftarrow Q \setminus \{u\}$
 9:         for each edge $(u, v, w)$ do
10:             if $dist[v] > dist[u] + w$ then
11:                 $dist[v] \leftarrow dist[u] + w$
12:     return $dist$
```

In the initialization phase, we would like to partition the nodes for our workers, so that each node has exactly one worker assigned to it. Let *assigned*(u) be a function which returns the worker to whom node u was assigned. Whenever line 7 is executed, each of the workers would calculate minimal distance vertex independently, and afterwards they would communicate to find the global minimal distance vertex.

There is a subtlety here: when a successful relaxation of edge (u, v, w) is performed in line 11, the worker assigned(v) must be made aware of this change. This can be resolved either by sending a message from assigned(u) to assigned(v), or by assigned(v) periodically scanning through distances of vertices it is responsible for rather than keeping them in a priority queue. Let's summarize the approach in a pseudocode.

Algorithm 4 Simple Parallel Dijkstra

```
 1: procedure SIMPLEPARALLELDIJKSTRA(G, SOURCE)
 2:     for $node \leftarrow 1$ to $N$ do
 3:         $dist[node] \leftarrow \infty$
 4:     $dist[source] \leftarrow 0$
 5:     assign_nodes_to_workers()
 6:     $Q \leftarrow G.nodes()$
 7:     while not Q.empty() do
 8:         for each core $c$ concurrently do
 9:             $u_c \leftarrow$ node in $Q$ such that $assigned(u_c) = c$ with minimal $dist[]$
10:         $u \leftarrow u_c$ with minimal $dist[]$
11:         $Q \leftarrow Q \setminus \{u\}$
12:         for each edge $(u, v, w)$ do
13:             if $dist[v] > dist[u] + w$ then
14:                 $dist[v] \leftarrow dist[u] + w$
15:                 communicate_change(assigned(v), v, dist[u] + w)
16:     return $dist$
```

Crauser et al. (1998) [5] introduced a strong improvement. In the paper *'A Parallelization of Dijkstra's Shortest Path Algorithm'* the authors describe another opportunity for exploiting parallelism. The idea is to run the algorithm in phases, and in each phase try to remove from Q many vertices whose distance is never going to change.[3]

Let $OUT(u)$ be the minimal weight of an edge going out of u. If $dist[u]$ is guaranteed not to change, all vertices v satisfying $dist[v] \leq dist[u] + OUT(u)$ can never be relaxed by u. At the beginning of each phase, we can calculate $L = min(dist[u] + OUT(u))$ among vertices u present in Q. Then, all vertices u satisfying $dist[u] \leq L$ can be removed from Q simultaneously![4]

Crauser et al. (1998)[5] prove that this optimization reduces with high probability the number of phases to $\sqrt{|V|}$ while staying work-efficient. In section 4 of their paper, the parallel implementation is also suggested. I leave the details to the *Implementation* chapter.

2.3.3 Shortest Path Faster Algorithm

Although one can extract a lot of parallelism from such a highly sequential algorithm like Dijkstra's, it may be better to design an algorithm with parallelism in mind.

Two most limiting factors of Dijkstra's algorithm are: the strong constraint that a vertex may never be relaxed again, and maintaining a priority queue. A relatively unknown algorithm which originated in the 1950s[5] lacks these bottlenecks. A simple version of this algorithm abandons priority in the queue and performs relaxations as long as there are some possible.

Algorithm 5 Chaotic relaxation

```
procedure CHAOTIC RELAXATION(G, SOURCE)
    for $node \leftarrow 1$ to $N$ do
        $dist[node] \leftarrow \infty$
    $dist[source] \leftarrow 0$
    $Q \leftarrow \{source\}$                    ▷ Q is a normal queue without priorities
    while not Q.empty() do
        $u \leftarrow Q.pop\_front()$
        for each edge $(u, v, w)$ do
            if $dist[v] > dist[u] + w$ then
                $dist[v] \leftarrow dist[u] + w$
                $Q.push(v)$
    return $dist$
```

[3] unlike just one in the standard algorithm

[4] As L is $min(dist[u] + OUT(u))$ over all nodes in Q, the above inequality holds for all nodes v such that $dist[v] \leq v$ and no node can relax them any further

[5] According to Wikipedia, the exact description (with a name algorithm D) can be found in: Moore, Edward F. (1959). 'The shortest path through a maze'. Proceedings of the International Symposium on the Theory of Switching. Harvard University Press. pp. 285–292. I have not managed to get access to this resource and verify this information myself.

Correctness of the algorithm is obvious - the relaxations are going to be performed as long as there is one. SPFA adds a tiny change in line 11: it does not add a vertex to the queue if it is there already.

Algorithm 6 SPFA

```
 1: procedure SPFA(G, SOURCE)
 2:     for $node \leftarrow 1$ to $N$ do
 3:         $dist[node] \leftarrow \infty$
 4:     $dist[source] \leftarrow 0$
 5:     $Q \leftarrow \{source\}$                    ▷ Q is a normal queue without priorities
 6:     while not Q.empty() do
 7:         $u \leftarrow Q.pop\_front()$
 8:         for each edge $(u, v, w)$ do
 9:             if $dist[v] > dist[u] + w$ then
10:                 $dist[v] \leftarrow dist[u] + w$
11:                 if $v$ is not in $Q$ then       ▷ The only change from ChaoticRelaxation
12:                     $Q.push(v)$
13:     return $dist$
```

This tiny change gives an empirically fast algorithm, which is believed to work in $O(|E|)$ on random graphs. In recent years, SPFA has been popularized by fans of algorithmic competitions, and it even has a Wikipedia page[6], but sources there appear to be several niche websites citing one another, and one academic paper, available in Chinese only. Analyzing mathematically the expected runtime of SPFA on random graphs is outside the scope of this project. Nonetheless, before the project I have carried out experiments to check behaviour of this algorithm on a large graph - California Road Network collected by various researches and introduced in [9]. Nodes were put into the queue 7 times on average, and the runtime was faster than Dijkstra's algorithm's. The initial experiments showed that not only was SPFA fast, but also promising in terms of parallel execution. This is because there are little restrictions on the order in which relaxations should be performed, which makes it suitable for parallel implementation.

Devising a parallel version is very natural, and uses a similar mapping of graph nodes to cores as parallel Dijkstra.

There are several concurrency issues possible: Q needs to be accessed by multiple workers, and collaboration on shared array *dist* is required. The discussion is continued in the *Implementation* chapter.

2.4 Reconstruction of paths

In section 2.3 the algorithms calculate only the array of shortest distances, rather than all the paths. This simplifies the implementations and explanations. To give a full picture,

[6] https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm

Algorithm 7 Parallel SPFA

```
procedure Parallel SPFA(G, source)
    for $node \leftarrow 1$ to $N$ do
        $dist[node] \leftarrow \infty$
    $dist[source] \leftarrow 0$
    $assign\_nodes\_to\_workers()$
    $Q \leftarrow \{source\}$                    ▷ Q is a normal queue without priorities
    while not Q.empty() do
        for each core $c$ concurrently do
            $u_c \leftarrow Q.pop\_front()$
            for each edge $(u_c, v, w)$ do
                if $dist[v] > dist[u_c] + w$ then
                    $dist[v] \leftarrow dist[u_c] + w$
                    if $v$ is not in $Q$ then
                        $Q.push(v)$
    return $dist$
```

this short section explains how to extend the algorithms to answer queries about the paths.

We introduce a predecessor array *pred*[] which will maintain previous node on the shortest path. For Dijkstra's algorithm and SPFA it is sufficient to update *pred*[] alongside with relaxations: when we are applying an update $dist[v] = dist[u] + w$, we also set $pred[v] = u$.

Contrary to the other two algorithms, in MatMul we can easily read the path between the source and destination in the right order by switching *pred* to a successor matrix *succ*. The change in the algorithm is marginal, too.

For initialization: $\underset{a,b}{\forall}\ succ[a][b] = b$. For the second phase, $min(newdist[a][b], dist[a][c] + dist[c][b])$ can be rewritten equivalently, after which it's clear when to update the successor matrix:

```
if $newdist[a][b] > dist[a][c] + dist[c][b]$ then
    $newdist[a][b] \leftarrow dist[a][c] + dist[c][b]$
    $succ[a][b] \leftarrow succ[a][c]$
```

Starting this way would not allow me to draw a connection betwen the algorithm and matrix multiplication so clearly.

There is another, even easier approach which can be useful especially if tight memory constraints are an issue: calculating the predecessor (or successor) on the spot, just based on the distance array, as in Algorithm 8. Once line 9 is reached, *pred* cannot be *None*, as there exists a previous vertex on the shortest path, and that vertex satisfies $dist[u] + w = dist[dest]$. This concludes the remark that with little additional effort, we can answer any shortest path query given the array of distances.

Algorithm 8 Reconstruction of paths

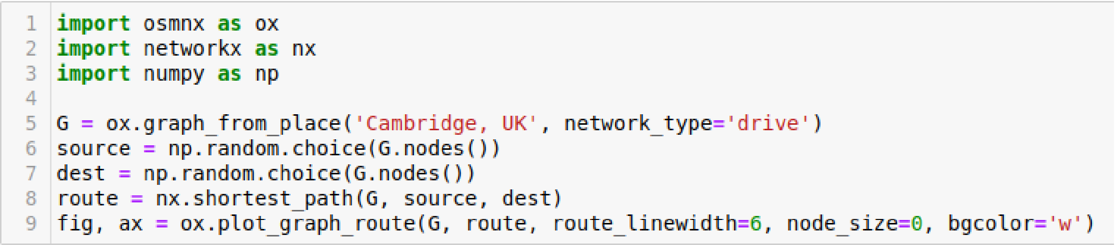
```
procedure Reconstruct-Path(source, dest)
    if source = dest then
        return [dest]
    else
        $pred \leftarrow None$
        for each edge ($u$, $dest$, $w$) do
            if $dist[u] + w = dist[dest]$ then
                $pred \leftarrow u$
        return append(Reconstruct-Path(source, pred), dest)
```

2.5 Datasets

To test shortest path algorithms, one needs graphs. I have decided to get my data with the help of OSMNX[2][7]. This python package, created by Geoff Boeing, is an excellent tool for working with road networks. It is a perfect fit for my project, because it provides one-line functions for:

- downloading road networks of any size based on geographical locations or names
- plotting networks or shortest paths
- automatic simplification of network topologies by removing graph nodes with 2 neighbors

Let's finish the chapter with a demonstration of OSMNX's usefulness. Creating the plot below took me only a couple of minutes.



```python
import osmnx as ox
import networkx as nx
import numpy as np

G = ox.graph_from_place('Cambridge, UK', network_type='drive')
source = np.random.choice(G.nodes())
dest = np.random.choice(G.nodes())
route = nx.shortest_path(G, source, dest)
fig, ax = ox.plot_graph_route(G, route, route_linewidth=6, node_size=0, bgcolor='w')
```

Figure (2.1) Code in jupyter notebook used to generate the plot below

[7] Although the author does not mention it explicitly, the name comes probably from crossing OpenStreetMap with NetworkX, which on its own explains a lot about this library

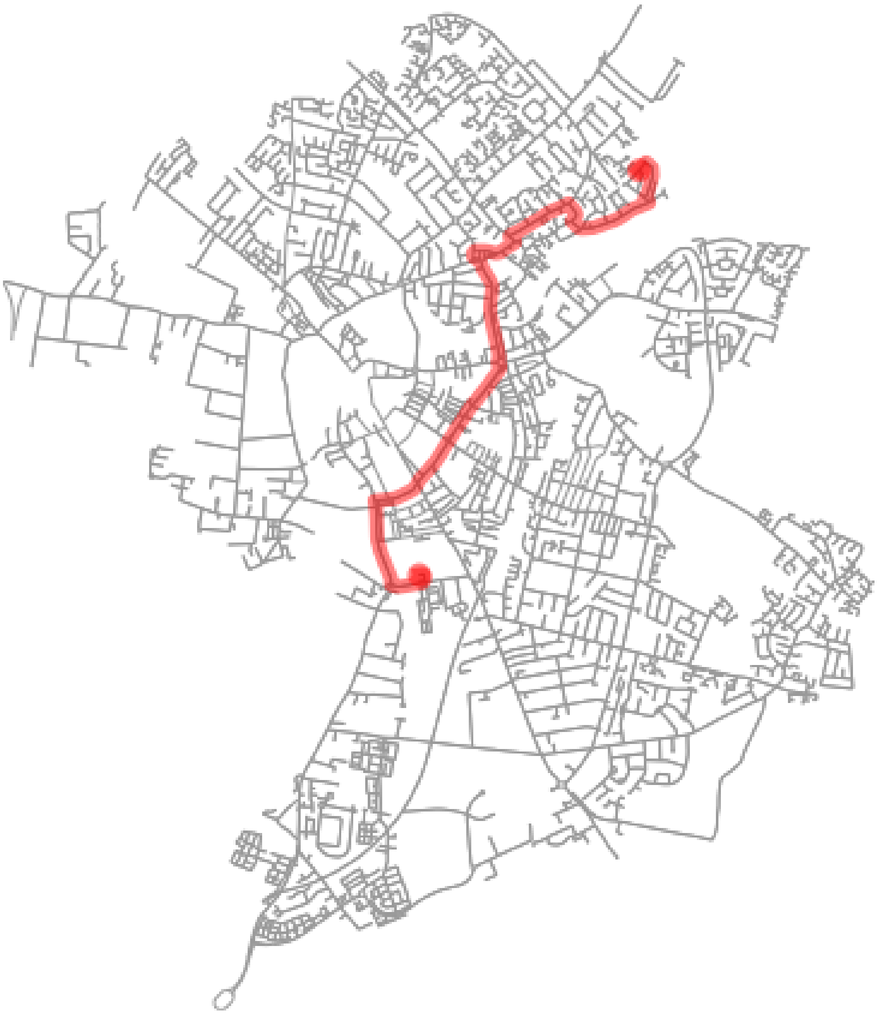


Figure (2.2) Cambridge road network with a shortest path between two randomly chosen junctions

Chapter 3

Implementation

3.1 Software Development - practices and tools

My project consisted of two parts - developing the simulator to run the algorithms, and coding the actual parallel algorithms in the simulator. I consider my work to have been done in a spiral-like method of software development. Although I had tried to carefully design the simulator by anticipating features needed by the algorithms, there were a couple of bits which needed modifications. Additionally, algorithms themselves allowed incremental improvements - initially do a part of code in a suboptimal, but easier way, then fix the shortcomings in the later versions.

The natural choice for implementing the project was Java. The language has excellent support for writing concurrent (I have used *java.util.concurrent* extensively throughout the project) and object-oriented programs. It also had the advantage of being taught in Part IA and IB courses, so I didn't need to spend time learning a new language for the project.

But Java also has drawbacks, and I couldn't imagine doing evaluation in Java. The results of running the algorithms were produced in the form of CSV files, so that I could then import them in Python easily. Python was a much better choice for doing data manipulations, comparisons and plotting. I had to use Python anyway if I wanted to make use of the aforementioned OSMnx module (I don't think that its ports to other languages exist). Altogether, I have written the entire project in Java with the exception of two Python notebooks: *graph_download.ipynb* and *evaluation.ipynb*. I have also made heavy use of the JetBrains IDEs (IntelliJ and PyCharm). I especially appreciated their capabilities for automatically resolving dependencies, imports and refactorings with usage search. Obeying builtin linter suggestions made me confident that the code follows good stylistic conventions.

Throughout the project I have been meeting regularly with my supervisors. Usually had a meeting once a week with either Dr Modi (Tuesdays), or Dr Greaves (Wednesdays). Additionally, during Michealmas and Lent terms my Directors of Studies held short (10 minutes) update meetings, where I would present recent developments in the project. When doing the work itself, I tried to follow the division suggested in my project proposal - biweekly sprints, but a few changes arose. The simulator work took longer than I had anticipated (partially because I made it much more sophisticated than originally planned), but later I caught up as a better simulator allowed quicker implementation of algorithms. I have also switched the order of implementing parallel Dijkstra and MatMul algorithms, as the latter seemed less error-prone.

I have also made use of tools with version-control systems to ensure that nothing gets lost. The code written in my GitHub repository was frequently committed and pushed, so that the changes would persist even in case of hardware failures. For writing my dissertation I have used Overleaf - a free online LaTeXeditor which allows real-time preview and collaboration on the documents, annotating the text with pop-up comments and even features a version control system. All of these features turned out to be very useful, especially in the unusual case of remote supervisions. Although I have been using the tool for many years and I generally trust Overleaf's cloud, writing the dissertation is a good occasion to become paranoid - so I have been doing additional backups daily on my machine and pushing the latest version to GitHub to be 101% sure of not losing substantial amounts of work.

3.2 Repository overview

Figure 3.1 presents a directory structure of my project. Crucially, the diagram mentions only example classes and files. I have followed a modular approach where classes have single responsibilities, as a result there are many different files. The diagram is much clearer if I include just example files. A full diagram is provided as Appendix A. Every piece of the source code present in that diagram was written by me, except for `FibonacciHeap.java` whose author is Keith Schwarz. The author permitted the usage of their work in other projects[1].

The repository consists of three main directories: `simulator/`, `graphs/` and `algorithms/` logically separating the three different components of the project. Each of them is further subdivided into directories, as shown in figure 3.1. Throughout the chapter I am describing what various building blocks are, what assumptions they take, and how they interact with each other.

3.3 Simulator

3.3.1 Overview

The simulator provides an evaluation environment to run the algorithms. In this section I am describing its internal implementation, as well as the functions it provides for implementing the algorithms.

The algorithms are expressed in threaded code[2] - they consist almost exclusively of calls to subroutines provided. I considered writing a byte code representation of the algorithms to be executed by the processor element models instead, but I realised that a

[1] Quoting directly from the author's website: *If you're interested in using any of this code in your applications, feel free to do so! You don't need to cite me or this website as a source, though I would appreciate it if you did.'*. More details can be found at: https://keithschwarz.com/interesting/

[2] not to be confused with multithreading

Figure (3.1) Repository overview diagram

```
route-planning
├── data
│   ├── cambridge.graphml
│   └── cambridge.txt
├── src
│   ├── algorithms
│   │   ├── APSP
│   │   │   └── MatMul
│   │   │       └── CannonMatMul.java
│   │   └── SSSP
│   │       ├── SSSPAlgorithm.java
│   │       ├── Dijkstra
│   │       │   ├── Dijkstra.java
│   │       │   └── DijkstraCore.java
│   │       └── SPFA
│   │           ├── SPFA.java
│   │           └── SPFACore.java
│   ├── graphs
│   │   ├── graph_download.ipynb
│   │   └── Graph.java
│   └── simulator
│       ├── internal
│       │   ├── TaskRunner.java
│       │   └── Scheduler.java
│       └── utils
│           ├── Message.java
│           ├── EvaluationEnvironment.java
│           └── LocalArray.java
└── test
    ├── DijkstraMultiThreadedTests.java
    └── evaluation.ipynb
```

threaded code implementation was going to more efficient. This also avoided having to write a compiler. The essence of threaded code is that we don't model a program counter explicitly in the simulator. Instead, the program counter of the simulating core is directly used (this was a Java thread in my implementation).

Looking from a very high-level perspective, a parallel system consists of cores executing various operations provided. Each core has access to its own local memory. One assumption taken is that local memories are insufficient to hold the entire graphs, but the graphs fit the total memory of the cores (don't need to be streamed from some external storage). Therefore, the cores may need to communicate their data to other cores. They can do it through message–passing.

3.3.2 Tasks and Phases

Since the early days of computing, algorithms are expressed as sequences of phases in which certain tasks are done. In the simulator `Tasks` are classes for chunks of work to be done. They have a method `execute()` which allows to specify the operations to be done for the core, as in the example. It uses the provided simple matrix multiplication subroutine to multiply two matrices stored in local memory of some core.

```
Task multiply_matrices = new Task(myCore) {
    @Override
    public void execute() {
        myCore.simpleMult(myCore.c, myCore.a, myCore.b);
    }
};
```

`Tasks` are grouped into `Phases`. Under the hood, `Phase` class has a thread-safe queue of Tasks and almost nothing more.

Every sensible system should provide guarantees when effects of certain operations are visible - it's unreasonable to assume they occur immediately. I have chosen the bulk-synchronous model: the tasks can assume that effects of Tasks from previous phases have already occurred. They cannot assume whether effects of Tasks from the same phase have occurred or not. Apart from the messages being passed, the cores run in isolation. An example consequence is that when cores need to pass some data cyclically, sending the data should be done in one phase and receiving the data in the next phase.

Additionally, there is a functionality for a task to be repeated. This feature is added to support types of algorithms which operate until there are no changes.

One subtlety arose in the design process. The single-threaded debug mode would deadlock if any of the Tasks contained operations that were blocking. I have investigated the possibility of using a Java package for coroutines to allow the execution to be suspended and resumed, but decided that adding it would be overly complicated for the needs and stuck to the assumption that Tasks execute non-blocking methods which run to completion (but they may, of course, in parallel with other tasks). In hindsight, this design decision did not turn out constraining.

3.3.3 Scheduler and task runners

The important requirement was that the algorithms are described in terms of operations on abstract cores. `Scheduler` and `TaskRunners` are responsible of handling that and running the computation on the physical machine. The `TaskRunner` class extends a Java Thread and provides a way to run tasks from phases. `Scheduler` spawns these runners and manages them to ensure that the environment does not deadlock. Once all the tasks from a phase are executed, the `Scheduler` instructs the runners to finish polling new tasks from this phase and switch to the next one instead (or issues the command for runners to terminate).

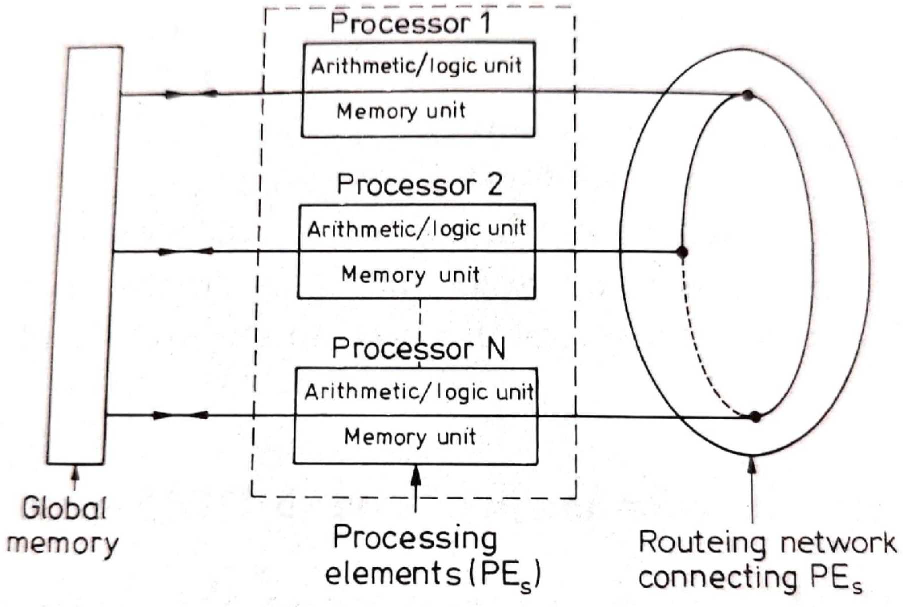
The asynchronous execution posed some challenges. Runners need to poll new tasks from the phase in a non-blocking way. Initially I just borrowed my own implementation of a queue done in practical 3 of FurtherJava course last year, but ultimately I've changed it to a ConcurrentLinkedQueue from java.util.concurrent package, as it had the desired behaviour, too. Once all the runners receive null tasks in the same phase, the phase is finished.

The operation of scheduler and task runners follows closely the thread pool design pattern. Summing up, the implementation is hidden in simulator/internal; the programmer just needs to specify an appropriate number of threads. For my machine the optimal number is 8, but better computers or cloud facilities may benefit from using more threads.

3.3.4 Cores and processor architectures

In the simulator, cores are the entities performing the computations and communicating with each other. Depending on the system, the processing elements may be extremely simple, capable of only doing basic bit manipulations, or as powerful as a modern computer. My design principle was not to constrain the programming of algorithms too much. Talking in terms of Flynn's taxonomy[7], I was aiming to support the MIMD design: each processing element operates asynchronously and independently of others, though might communicate with others via a network of interconnections.

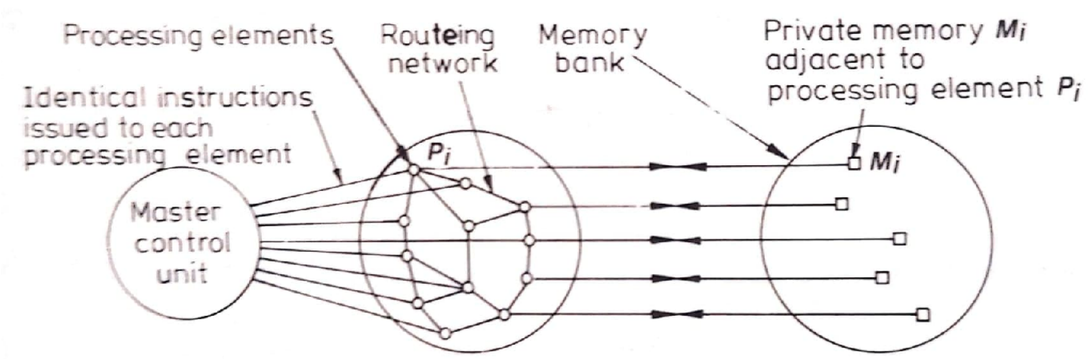
Figure (3.2) MIMD design. The diagram's source is Dr Modi's book *Parallel algorithms and matrix computation* [10]. I would like to thank the author for granting me permission to use it.



Tweaking this to support somewhat simpler SIMD architectures would be easy: one could specify tasks so that each core does the same simple operation.

In both designs a MasterCore may appear, though in SIMD ones it is essential. In my design, the MasterCore has the same capabilities as a normal core, but is serves a different purpose in the algorithms. It is useful especially for doing initialization and termination work, as well as some bookkeeping.

Figure (3.3) SIMD design. The diagram's source is Dr Modi's book *Parallel algorithms and matrix computation* [10]. I would like to thank the author for granting me permission to use it.


Processing elements
Routeing network
Memory bank
Private memory M_i adjacent to processing element P_i
Identical instructions issued to each processing element
P_i
M_i
Master control unit

Listing (3.1) MasterCore.java

```java
package simulator.utils;
public class MasterCore extends Core {
    public MasterCore(EvaluationEnvironment env, int coreID) {
        super(env, coreID);
    }
}
```

The Core class provides primitives mentioned in the requirements analysis with a few additional ones: operations for primitives, naive multiplication of matrices, as well as sending and receiving messages from other cores. I have also added utilities for data structures residing in local memory of the cores: arrays and heaps. Each operation is tracked and measured by the evaluation environment. I have decided not to reinvent the wheel and subclassed the standard Java collections. This guaranteed no bugs in their behaviour. The instrumentation to find the communication overhead adds only a simple call to the simulator's tracking class[3]. This way I have achieved the goal of simplicity: the programmer can write nearly the same code as they would normally write in Java. The only exception is that operations done by cores need to be wrapped in the execute() method of class Task.

Processor architecture class provides methods to specify the network of interconnections between the cores. This is important as far as communication is concerned, as processing elements communicate along these connections. To reflect that indirect message-passing might take place as well, although a message following a non-direct path of course needs more time than using a direct connection. The distance between the cores is assumed to be the length of the shortest path in the graph of interconnections and used to calculate the cost of propagating the messages.

[3] More about tracking and costs of operations in the section about Estimators

3.3.5 Message passing

The cores have methods to send and receive messages. The behaviour of these is implemented in the `CommunicationHandler` and `SingleConnectionHandler` classes. The former manages all possible connections, and uses a `ConcurrentHashMap` to guarantee thread-safety and efficiency. The latter uses a `ConcurrentLinkedQueue` which implements a simple producer-consumer relationship. It assumes that messages sent between the same pair (source, destination) are delivered in order. The `Message` itself can be arbitrary - it contains a list of objects. Receiving a message is a blocking operation, and trying to receive a message sent by another node in the same phase would be a violation of the assumptions.

Certain algorithms may use some sort of hardware broadcast, and I wanted to support this somehow. I have decided to create a method `setParallelMode()` which adjusts the measurements to behave as though the operations were achieved by parallel hardware.

3.3.6 Tracking operations and their costs

During the execution, we measure how much time has elapsed on each of the cores since the start. This is not the wall clock from the execution on the physical machine; it increases asynchronously throughout the simulation according to the operations the workers perform.

As a rule of thumb, whenever some operation is done by one of the cores, it is being tracked by an object of class `Tracker`, which takes appropriate action. There are two cases in which synchronizing the times may be needed: upon receiving a message and upon ending a phase. Receiving a message send at time t_0 whose propagation delay was t_1 is not possible before the time t_0+t_1. Ending a Phase may serve as a thread barrier - the time on each core is set to the maximum among all cores. As chaotic relaxation algorithms benefit from relaxing this constraint, the synchronization after a phase is optional.

Various costs associated with operations are estimated by an object of the `Estimator` class. In the default implementation memory access & ALU operation latency, message processing and propagation delays, and data structure operation costs need to be estimated.

The last one of these decreases the granularity and can be seen as an unsatisfactory assumption. However, removing it would require implementing the data structures with the primitives provided rather than instrumenting Java implementations. The implementations and their efficiencies may differ; it's easier to change the estimation to a better one than need to reimplement the data structures. Forcing the potential user to change the behaviour of simulator's internals just to get different measurements would also violate principles of good software design, hence I did not choose this option.

The default estimations are discussed in the *Evaluation* chapter, as they belong there more naturally. The goal of my implementation was to provide ways of measuring different circumstances easily, and this has been achieved.

3.3.7 Evaluation environment

EvaluationEnvironment class wraps everything up and helps the building blocks communicate by providing getters and setters. Many of them are hidden from the outside world. In fact, to have a working evaluation environment, one only needs to specify three things: number of threads of the physical machine to run the computation, the processor architecture and the estimator (or choose the default implementations). The code snippet from one of the test classes shows how to initialize an evaluation environment.

Listing (3.2) Setting up an evaluation environment

```
public void setUp(String method) {
        this.env = new EvaluationEnvironment();

        this.estimator = new Estimator(env); // a default estimator
        this.proc = new Lattice(env, 4, method); //processing architecure introduced in the next section

        this.env.init(this.proc, this.estimator, 8); //running on 8 threads
    }
```

Running a phase of the algorithm is even simpler: env.runPhase(). Collection of the results between the execution of phases (or at the end) can be done by accessing the tracker. By default, the interface for shortest path algorithms returns an instance of Metrics class which contains the evaluation results: estimates of time to finish the execution and how much work was performed in total. It is easy to extend this beyond the goals of my project: for other algorithms or if more granularity is required.

3.4 Parallel matrix multiplication

Section 2.3.1 has been concluded with the remark that it is sufficient to implement an efficient parallel matrix multiplication algorithm which works for square matrices. In this section I am describing the details of this procedure.

Our goal is to multiply two square matrices A and B, each containing N^2 elements. Consistently with usual notations, their product $C = A \cdot B$ satisfies $C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj}$.

Let's initially consider the simplified scenario in which we have exactly N^2 processing elements available. An ideal network for processing the matrices would be a two-dimensional mesh of these N^2 processing elements arranged in a grid, with vertical and horizontal interconnections. In the literature we can find it under different names: lattice, $2D$-torus or grid interconnect.

Toroidal interconnections have been of interest for years. They are a natural choice for many matrix-processing algorithms, have low latency and energy consumption. Their disadvantages include difficult wiring of wrap-around connections and costly communication between distant pairs of processing elements. Cannon's algorithm[3] for multiplying matrices with processing elements (PEs) works out nicely.

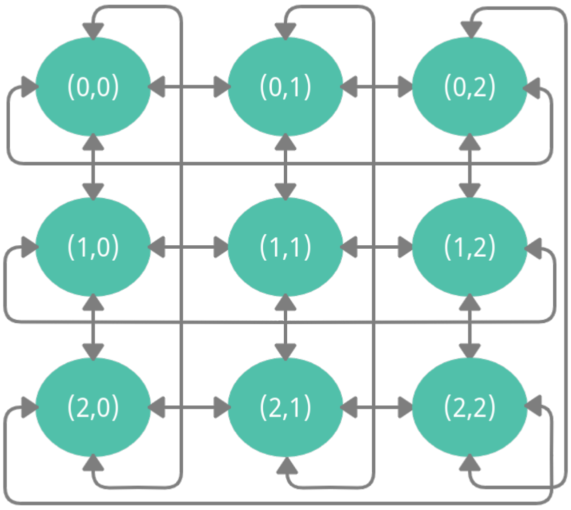


Figure (3.4) A bidirectional torus interconnection of 9 processing elements

The goal is to have the processing element PE_{ij} compute $C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj}$. It would be unreasonable to assume that the processing elements have efficient random access to the entire matrices. Instead, we need to cleverly move the data around.

The calculation will be done in N steps. In each of the steps, PE_{ij} will be adding $A_{ik} \cdot B_{kj}$ for some k. Crucially, the order of additions for different PEs will not be the same. The key is to use diagonals and permute them appropriately. A good diagram is worth a thousand words of explanation.

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix} \to \begin{bmatrix} 1 & 2 & 0 \\ 2 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \to \begin{bmatrix} 2 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 0 \end{bmatrix} \tag{3.1}$$

It is easy to see that PE_{ij} starts by doing the addition for $k = (i + j) \bmod N$, and between the steps the value increases by one (modulo N). The movement described above corresponds to this computation:

$$\begin{bmatrix} A_{00} \cdot B_{00} & A_{01} \cdot B_{11} & A_{02} \cdot B_{22} \\ A_{11} \cdot B_{10} & A_{12} \cdot B_{21} & A_{10} \cdot B_{02} \\ A_{22} \cdot B_{20} & A_{20} \cdot B_{01} & A_{21} \cdot B_{12} \end{bmatrix} \to \begin{bmatrix} A_{01} \cdot B_{10} & A_{02} \cdot B_{21} & A_{00} \cdot B_{02} \\ A_{12} \cdot B_{20} & A_{10} \cdot B_{01} & A_{11} \cdot B_{12} \\ A_{20} \cdot B_{00} & A_{21} \cdot B_{11} & A_{22} \cdot B_{22} \end{bmatrix} \to \begin{bmatrix} A_{02} \cdot B_{20} & A_{00} \cdot B_{01} & A_{01} \cdot B_{12} \\ A_{10} \cdot B_{00} & A_{11} \cdot B_{11} & A_{12} \cdot B_{22} \\ A_{21} \cdot B_{10} & A_{22} \cdot B_{21} & A_{20} \cdot B_{02} \end{bmatrix} \tag{3.2}$$

The movement of data is remarkably simple: between the iterations elements of matrix A move (cyclically) to the left, and the elements of matrix B move upwards. The pseudocode of the matrix multiplication algorithm is presented in Algorithm 9.

It turns out that if sizes of matrices (N^2) are larger than the number of processing elements available (let's call it P^2), the same idea still works. It is enough to slice the matrices into P^2 submatrices of size approximately $(\frac{N}{P})^2$, and we're done! To deal with edge cases nicely, I have enlarged matrices A and B with ∞ values so that N is divisible by P.

The pseudocode translates directly to the implementation running on the Evaluation Environment. It is implemented in `CannonMatMul.java`. The algorithm is simple to implement and efficient in terms of moving data around. Its greatest disadvantage is

Algorithm 9 Parallel Matrix Multiplication

```
procedure CannonMatMul(A, B, N)
    for each processing element $PE_{ij}$ concurrently do
        $k \leftarrow (i + j) \bmod N$
        $tempA_{ij} \leftarrow A_{ik}$
        $tempB_{ij} \leftarrow B_{kj}$
        $C_{ij} \leftarrow tempA_{ij} \cdot tempB_{ij}$
        for $step \leftarrow 1$ to $N - 1$ do
            send $tempA_{ij}$ to the left neighbour
            send $tempB_{ij}$ to the up neighbour
                            ▷ Reminder: send and receive should not be in the same phase!
            $tempA_{ij} \leftarrow$ item from the right neighbour
            $tempB_{ij} \leftarrow$ item from the down neighbour
            $C_{ij} \leftarrow C_{ij} + tempA_{ij} \cdot tempB_{ij}$
    return                                          ▷ $C_{ij}$ is now the desired result
```

reliance on the grid of processing elements being available - otherwise the movement of data may become quite inefficient.

3.5 Parallel Dijkstra's algorithm

This section extends the discussion in section 2.3.2 and is concerned with the implementation details of the described ideas. Let's recall the pseudocode of SimpleParallelDijkstra.

In line 5 there is a mysterious function assign_nodes_to_workers(). The authors of *'A parallelization of Dijkstra's shortest path algorithm'* [5] suggest a random initialization in order to prove that theoretical bounds occur with high probability. I have initially followed their description, but later decided to examine an improvement which aims to optimize the communication costs. Since it's an improvement to both Dijkstra's and SPFA algorithms, the description follows in section 3.7. For now, a mapping of nodes to workers is chosen uniformly at random for each node. The graph is then partitioned according to this mapping in a node-centric way: each worker knows about the subgraph induced by its nodes.

Instead of a maintaining a global priority queue, it is better when each of the workers maintains its own, local priority queue which is going to hold nodes assigned to it sorted by the current distance. This eliminates the need for implementing parallel data structures. Additionally, another priority queue is kept at each node to calculate minimum of the expression $dist[u] + OUT(u)$, where $OUT(u)$ was the minimal weight of an edge going out of u.

The main while loop of the parallel Dijkstra's algorithm works in four phases:

1. The cores communicate to find the threshold $L = min(dist[u] + OUT(u))$ among all nodes which are still in the queues. If a core has no nodes in their queue, it can terminate.

Algorithm 10 Simple Parallel Dijkstra

```
procedure SimpleParallelDijkstra(G, source)
    for $node \leftarrow 1$ to $N$ do
        $dist[node] \leftarrow \infty$
    $dist[source] \leftarrow 0$
    assign_nodes_to_workers()
    $Q \leftarrow G.nodes()$
    while not Q.empty() do
        for each core $c$ concurrently do
            $u_c \leftarrow$ node in $Q$ such that $assigned(u_c) = c$ with minimal $dist[]$
        $u \leftarrow u_c$ with minimal $dist[]$
        $Q \leftarrow Q \setminus \{u\}$
        for each edge $(u, v, w)$ do
            if $dist[v] > dist[u] + w$ then
                $dist[v] \leftarrow dist[u] + w$
                $\mathrm{communicate\_change(assigned(v), v, dist[u] + w)}$
    return $dist$
```

2. Each core removes all the nodes from its queues with the current distance not exceeding L. For each of the removed nodes, the final distance is now calculated.

3. For each node u of the removed nodes, all edges (u, v, w) should be considered for relaxation. Applying the relaxation operation guarantees that $dist[v]$ is at most $dist[u] + w$, which happens if u was the direct predecessor of w. The relaxations form a collection of requests of the form $(v, dist[u]+w)$ which signal a possible lower bound on the distance. The request (v, d) is then sent to the core *assigned(v)* in a batch.

4. Each core receives the batch of possible updates and processes them similarly to the traditional sequential algorithm: updates the distances and triggers `Q.decreaseKey()` on the relevant nodes.

The algorithm finishes once all the cores have terminated. Each core has computed shortest distances for the nodes assigned to it. The algorithm, unlike matrix multiplication, does not use any special assumptions about the interconnections between processing elements. However, using special properties of the available network may bring additional benefits.

3.6 Parallel SPFA implementation

This section is very similar to the previous section, as SPFA and Dijkstra's algorithm have a lot in common. Let's recall the proposed pseudocode of a parallel SPFA (Algorithm 11).

Algorithm 11 Parallel SPFA

```
 1: procedure Parallel SPFA(G, source)
 2:     for $node$ ← 1 to $N$ do
 3:         $dist[node] \leftarrow \infty$
 4:     $dist[source] \leftarrow 0$
 5:     assign_nodes_to_workers()
 6:     $Q \leftarrow \{source\}$                    ▷ Q is a normal queue without priorities
 7:     while not Q.empty() do
 8:         for each core $c$ concurrently do
 9:             $u_c \leftarrow Q.pop\_front()$
10:             for each edge ($u_c$, $v$, $w$) do
11:                 if $dist[v] > dist[u_c] + w$ then
12:                     $dist[v] \leftarrow dist[u_c] + w$
13:                     if $v$ is not in $Q$ then
14:                         $Q.push(v)$
15:     return $dist$
```

Similarly as before, the function in line 5 in the initialization phase partitions the graph G randomly. Q is no longer a priority queue, just a typical FIFO one. It was tempting to implement the algorithm directly without too many modifications - just make sure that Q and the arrays are safe to be accessed in parallel by multiple workers. This could create contention around the ends of the queue, as many workers would be trying to poll or add nodes at the same time. Although there are standard implementations of work stealing which use a mix of FIFO and LIFO queues implemented as far as possible with lock-free, atomic primitives (one of which is described by Blumofe and Leiserson in [1]), the more promising design seems to be the one in which each processing element operates on their own queue. Additionally, this way renders the request-generation system from Dijkstra's parallelization usable here, too.

There is at least one more benefit, but this one is a bit more subtle. Checking if v is in Q (line 13) can be done with a simple boolean array. If this array is global, the effect of operation updating distance in line 12 must be visible to all other cores quickly. If this wasn't the case, we would allow a tricky concurrency bug to occur. Let's assume that work of cores C_1 and C_2 happens in this unlucky order:

1. C_2 is in the process of relaxing from node v and gets $dist[v]$.

2. C_1 is in the process of relaxing from node u and sets $dist[v]$ to a new, smaller value.

3. C_1 checks that v is in the queue, so doesn't add it once again.

4. C_2 removes v from the queue and proceeds to relax some other node z with $dist[v] + length(v, z)$.

After the fourth step is executed, vertex v may never be put in the queue again. Although its distance is calculated correctly, it may not be calculated correctly for node z if the shortest path to z went through v.

If the queues and arrays are local to cores this problem disappears, as there is only one core capable of manipulating data about node v. This design looks superior to the other one, as not imposing sequential consistency rules may be crucial for performance of this chaotic relaxation algorithm. The cores can just operate asynchronously until they have no more changes to process.

3.7 Optimizing communication costs for SPFA and Dijkstra

Although choosing the mapping of nodes to processing elements in the function assign_nodes_to_workers() randomly is satisfactory, we could try to improve beyond random. If we assign entirely at random, then the processing elements need to send the messages all around the network of interconnections. It would be better if most of the graph's edges connected nodes assigned to the same processing element, or at least to a neighboring one.

In this section I am describing the optimization for lattice interconnections. It is not obvious that this is going to improve anything - although communication cost may decrease, utilization of processing elements may decrease, too. This is because many workers may be idling as their chunk hasn't started yet or has already finished.

Real-world road networks downloaded with OSMnx have GPS coordinates associated with them. I have written a short script in `graphs/graph_download.ipynb` to add those to the graph format. Using these coordinates, a simple partitioning is performed. Assuming there are P^2 processing elements, $P-1$ vertical and $P-1$ horizontal lines are added independently. Along each dimension, after roughly $\frac{N}{P}$ vertices a new line is added, as in the diagram below. Each of the P^2 formed regions contains nodes mapped to some processing element, as shown in figure 3.5. Usefulness of this modification is measured in the *Evaluation* chapter.

3.8 Testing the code

Parallel programs can be difficult to debug, as they can manifest subtle bugs in bizarre ways. In this section I am addressing the difficulties of testing the code for correctness. Performance evaluation follows in the next chapter.

Throughout the project I have been using test-driven development for testing the algorithms and the evaluation environment. I have prepared the graphs for test runs early on. Each of the algorithms was tested on six small road networks of New York city center. The smallest one had just 3 nodes and 2 edges and was useful for verifications on paper - if the issue was major[4], it would appear on a testcase easy to follow with a debugger The largest of the graphs had 27 nodes and 64 edges.

Each of the algorithms was used to calculate all pairs of shortest paths for these graphs and verified against the model answer. This model answer was computed with networkX's

[4] Example bug: forgetting to collect the results from processing elements

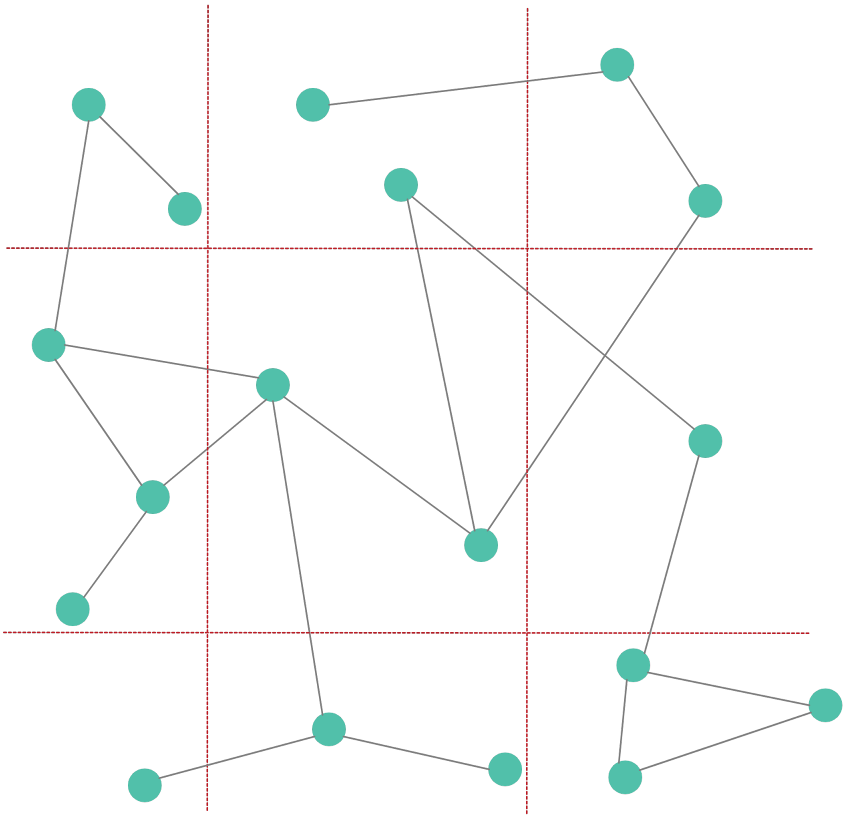


Figure (3.5) A graph mapped to 9 workers

`shortest_path_length()` function. To account for floating-point errors, results a and b were considered to be equal if $abs(a - b) < \epsilon$. The value of ϵ was chosen to be 10^{-3}.

Each test was run on the evaluation environment in a single-threaded and multi-threaded mode. If the two results differed, it was a sign of a concurrency bug. If even the deterministic single-threaded mode failed, the debugging task was much simpler, as the issue was reproducible. I could then easily detect the place where the mistake happened and fix it.

Altogether, this test set achieved 99% coverage of lines for the algorithms package[5], and 100% coverage of classes and methods. The same tests also cover a significant chunk of the simulator modules which achieve over 80% of coverage. This is partially because there are a few simple methods and functionalities which I had implemented, but ended up not being used in the implementation of the algorithms.

One more difficult challenge remained. The evaluation environment outputs a number as a measurement, how can we be sure that it is the appropriate one?

The modular approach taken helped me to verify this. I wrote various simple programs which performed the operations available in some order. If I wanted to verify that operation X was tracked and counted correctly, I could supply an Estimator which assessed the cost of operation X as 1, and the costs of all the other operations as 0. The result was then compared with the number of operations X the code performs. Doing such a test for all the simulator's primitives was a sanity check that the measurements provided

[5] calculated with the run-with-coverage feature of IntelliJ IDEA

are legitimate.

The tests, following the Arrange-Act-Assert design pattern, were implemented with the features of JUnit testing library. I have not set up a continuous integration pipeline, as clicking the IntelliJ `run` button seemed sufficient for the needs of this project.

Listing (3.3) Example code snippet from DijkstraMultiThreadedTests.java

```java
@BeforeEach
public void setUp() {
    this.env = new EvaluationEnvironment();

    this.estimator = new Estimator(env);
    this.proc = new Lattice(env, 4, "Dijkstra");

    this.env.init(this.proc, this.estimator, 8);
}

@Test
public void testDijkstraSmallTest0() {
    // Arrange
    Graph g = new Graph("data/test_0.txt");
    SSAlgorithm dijkstra = new SSAlgorithm(env, g, new Dijkstra(env))
        ;

    // Act
    Matrix output = dijkstra.getAllPairsShortestPaths(g).getKey();
    Matrix model_ans = new Matrix("data/test_0_ans.txt");

    // Assert
    Assertions.assertTrue(output.equalsWithEpsilon(model_ans, EPS));
}
```

Chapter 4

Evaluation

The project set out to explore three different approaches to the shortest path problem, implement the ideas associated with their proposed parallel adaptations. All four components have been delivered and thoroughly described in the dissertation – the evaluation environment, Matrix Multiplication, Dijkstra's and SPFA algorithms. They have been tested according to the methodology described in section 3.8. It remains to provide a comparison of their performance, which this chapter is aiming to achieve. At first, we are focusing on the All-Pairs Shortest Paths problem as a common ground, as MatMul algorithm does not provide means to solve the Single-Source Shortest Path problem alone. SSSP algorithms are more versatile, as they can be repeated for each possible source. Later, the comparison of the two SSSP algorithms on larger graph instances follows.

4.1 Theoretical complexity

Let's start with a theoretical estimation of algorithms' computational complexities. In the section I am analysing their big-O properties. As Rowell (2007) [8] has shown, standard big-O notation has difficulties with formality when dealing with multivariate analysis. Thus, to be precise formally, I am using their definitions which try to preserve the behaviour of the big-O notation.

4.1.1 Computation and communication costs

When evaluating parallel algorithms, we may differentiate between computation and communication costs. As number of processing elements increases, the time to run the computations may decrease, while the time the workers spend to communicate may increase even more rapidly. It is not true that increasing the number of workers always improves the results. In fact, this might worsen them, as the growth in communication costs outweighs potential benefits.

In the subsections below, I am assuming the conventions followed throughout the dissertation: N vertices and M edges in the graph, P processing elements available.

4.1.2 Matrix Multiplication

MatMul algorithm has the easiest complexity analysis. It is important to note that its behaviour is not influenced by graph's shape, as it operates on the graph's distance matrix. Thus, only N and P matter in the analysis.

The algorithm operates in $log(N)$ steps, in each step two NxN matrices are multiplied. This gives $O(N^3 \cdot log(N))$ operations altogether to be executed by P processing elements. Additionally, there is some communication cost associated with moving the data from matrices around: at each of the $\sqrt{P} \cdot log(N)$ iterations the chunks of the matrix are redistributed, which gives $O(N^3 logN)$ bytes in the messages altogether. Each core sends data and receives it a constant number of times in each iterations, which gives $O(\sqrt{P} \cdot log(N))$ per core. The computation cost is $O(\frac{N^3 log(N)}{P})$. When P is close to N^2, both these costs meet in the middle. However, this number of cores is only theoretically optimal. In reality, the constants associated with communication costs can be significantly larger than communication cost, and this makes the big-O analysis inaccurate.

4.1.3 Dijkstra's algorithm

Dijkstra's algorithm operates in phases. Each element is removed from the priority queue during exactly one of those, and sends the message to each neighbor. Before each phase the processing elements need to communicate in order to find the threshold $L = min(dist[u] + OUT(u))$. This adds $O(P)$ additional messages per phase. Assume there were C phases of the algorithm in total, and that n_1, n_2, ..., n_C vertices were removed from the queues. The communication cost is $O(M + C \cdot P)$, combining the two factors described above. Total amount of work performed across all the cores is $O(M + N \cdot log(N))$, but the utilization of processing elements may differ greatly depending on n_1, n_2, ..., n_C. Hence, it is difficult to provide estimates of the cost per core, unlike in the matrix multiplication case.

Crauser et al. prove in [5] that for random graphs with uniform edge weights from $[0; 1)$, C is $O(\sqrt{N})$ with high probability. In my experiments I have empirically verified that this does not hold for road networks: the average number of phases was between $\frac{n}{4}$ and $\frac{n}{3}$ across all my tests.

4.1.4 Shortest Path Faster Algorithm

Analyzing even the single–core version of SPFA poses a challenge. I have been able to find only partial reports of empirical studies which claim that on average SPFA has $O(k \cdot M)$ complexity where k is a small constant. Worst–case behaviour of $O(N \cdot M)$ is rarely seen in practice, even though one can construct graphs on which SPFA performs particularly poorly.

When P processors are available, the analysis is even harder and I do not have a definite answer to this question. For moderate P it offers great utilization of processing elements, as they can do useful work as long as their relaxation queues are not empty. This is hard to quantify mathematically; in this case we should proceed to discussing the empirical results.

4.2 Methodology: estimations used

In section 3.3.7 I've mentioned that costs of various operations are estimated. The default estimator is a Gaussian estimator which samples the distributions from $\mathcal{N}(\mu, \sigma^2)$. The following default estimations are used:

- Local memory operations: $\mu = 0.5$ nanoseconds (ns).
- Binary ALU operations: $\mu = 1$ ns (corresponding roughly to two reads)
- Message processing operations: $\mu = 3$ ns per packet for both sender and receiver.
- Message propagation delay: $\mu = 30$, then the value is multiplied by distance between the sender and receiver.[1]
- Priority queue operations: $\mu = 10$. This value is chosen to represent the logarithmic factor and large constants which can arise from implementations.

For each operation, the standard deviation σ is chosen to be $\frac{\mu}{5}$. This helps to bound the standard deviation of the overall result trivially: it should not be greater than 20%. We should also note that the result is a summation of many measurements, so the Central Limit Theorem [4] applies. As the number of operations OP grows large, the standard deviation is not multiplied by OP as a simple analysis would suggest, but by $\sqrt{OP}$ instead. Practically speaking, this means that for the numbers of operations which can be performed by modern computers, the relative standard deviation of the result is significantly less than its 1%. In section 4.6 I am investigating how the parameter choices affect the results.

4.3 Execution times

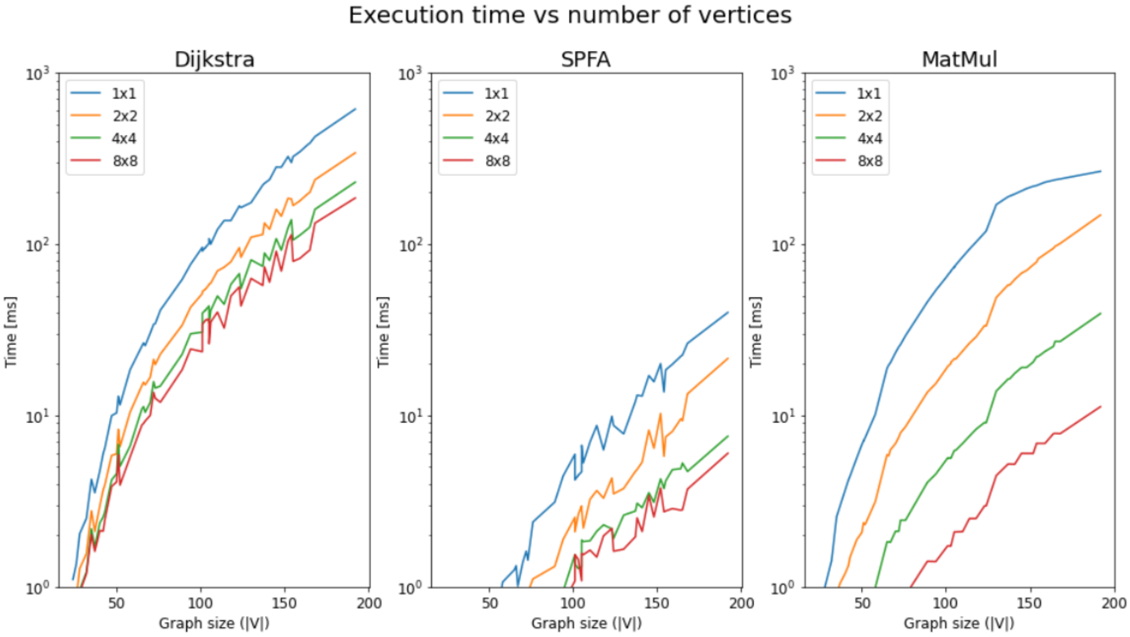
I have evaluated the three implemented algorithms using 50 road networks of various sizes between 20 and 200. The road networks downloaded in `graph_download.ipynb` are from the cities of New York, Cambridge, Berlin, Warsaw, Palo Alto and Rio de Janeiro to cover a broad range of urbanist trends. Dijkstra's algorithm fails to perform on par with

Figure (4.1) Comparison of various road networks

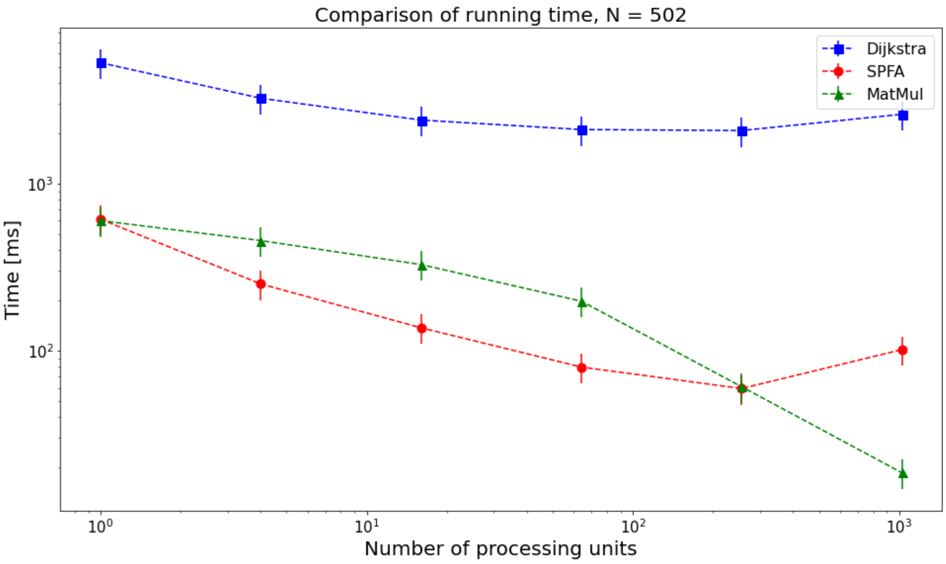
(a) Manhattan (b) Cambridge (c) Warsaw

[1] measured as number of conections on the shortest path between those

the other two algorithms, which forced me to use a logarithmic scale on y–axis. Lines correspond to the result obtained when lattices of PxP processing elements were used. The more processing elements, the better the execution time is, especially in the case of MatMul. This aligns with its theoretical analysis (offering good parallel speedups).



Let's investigate further and look closely at the results for one graph of bigger size. I have chosen $N \approx 500$ as for such N evaluating an algorithm of complexity $O(N^3 \cdot log(N))$ [2] posed a challenge, but was still possible. The computations to produce the data for the plot below took two hours on my machine. The error bars represent plausible standard deviations of 20%. Standard deviations calculated with the better approximation from Central Limit Theorem would be too small to be visible. In section 4.6 I am describing empirical calculations of these standard deviations.

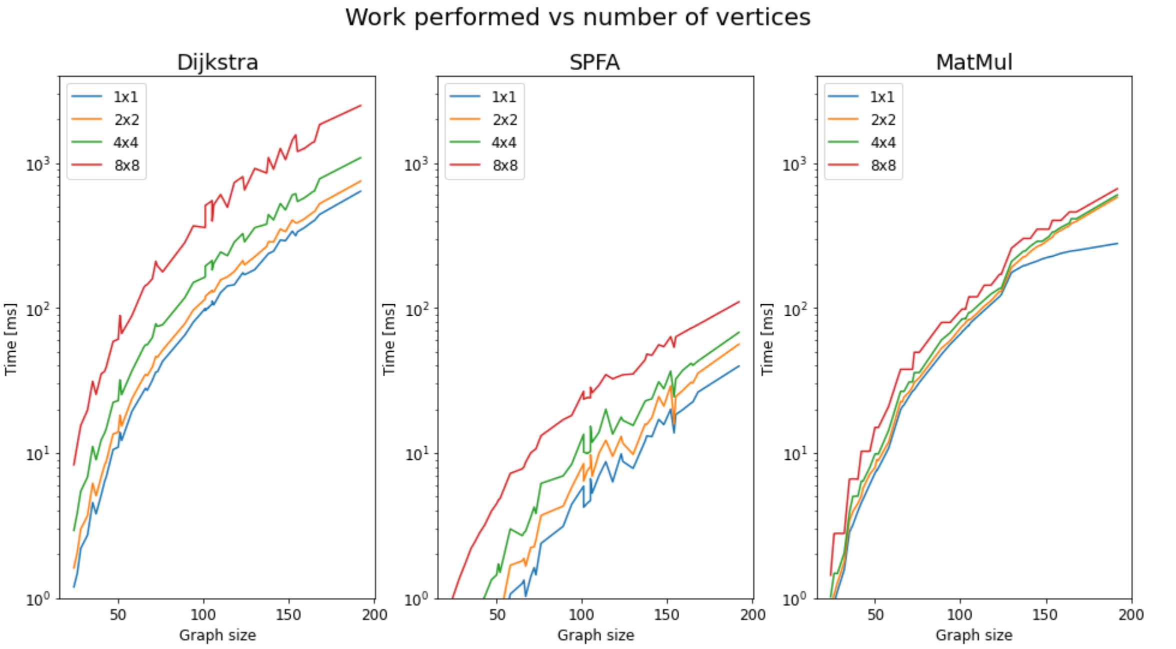


[2] with a huge constant factor arising from the simulator's overheads

The trend is clear, parallelization of Dijkstra's algorithm does not perform comparably well. SPFA offers a good parallelization, but inevitably exhibits a diminishing returns phenomenon. The graph's diameter poses a lower bound, as it needs to be processed sequentially. That diameter can be quite large in real–world road networks, which may contribute to the lack of improvement for greater values of P. Increasing the number of available processing elements would only decrease the execution time as communication cost would increase. Altogether, I would recommend implementing MatMul only if there is a need to compute all-pairs shortest paths and there is a large grid of processing elements available. In all other cases SPFA looks like a top choice.

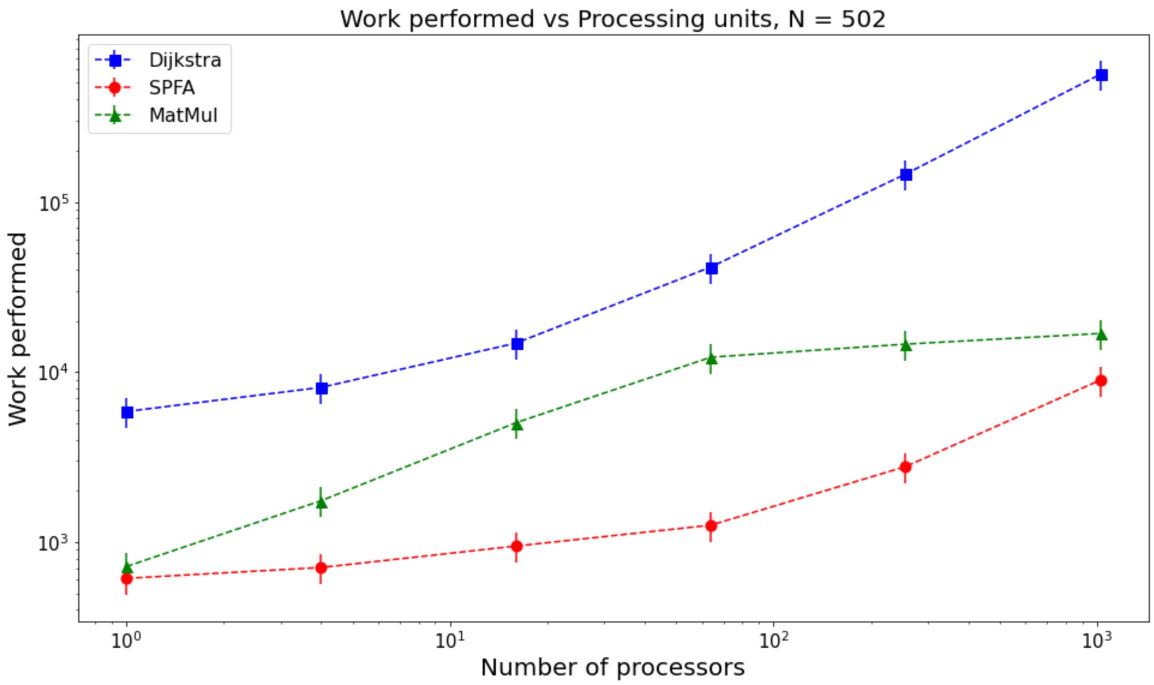
4.4 Energy efficiency

Although speed of algorithms is important, it is also important that they do not require too much energy. All three algorithms achieve the same results, but do so performing different amount of work overall. Let's look at the relative merits. The data for the plot below is collected by summing the estimates of work performed by cores rather than taking the maximum value among the cores. These include the costs associated with passing a message around, they evenly contribute to the sender's and receiver's work.



Interestingly, the lines have reversed their order now: the 8x8 processing grids achieve the shortest execution times, but on the other hand require the largest amount of work. This represents a tradeoff in choosing the right number of processing elements: one might prefer to reduce the number of cores used to get a slightly slower algorithm which saves the processing elements some of the work.

It looks as though SPFA was the most work–efficient of all three algorithms used. This result is slightly surprising to me – I was expecting parallel Dijkstra's algorithm to perform less work overall, as it never relaxes a vertex twice.



4.5 Investigating the optimization for communication costs

In section 3.7 I have defined an optimization which aims to improve the throughput and minimize communication costs. Before claiming any credit I need to quantify whether this constituted an improvement indeed. The plots describing the behaviour are voluminous, so I have placed them in Appendix B. The main insight from them is that the algorithms perform quite similarly, regardless of whether the optimization was used or not. There is also some noticeable improvement, but not in all scenarios. Is it sufficient to be statistically significant?

I have formulated a Null Hypothesis that the non–speedup versions perform no worse than the supposedly better versions of the algorithms. I have compared the results reported on the same test instances and found the following p–values on the binomial statistical test:

	Dijkstra	SPFA
Execution time	0.6382	0.0001
Work performed	0.0002	0.0000

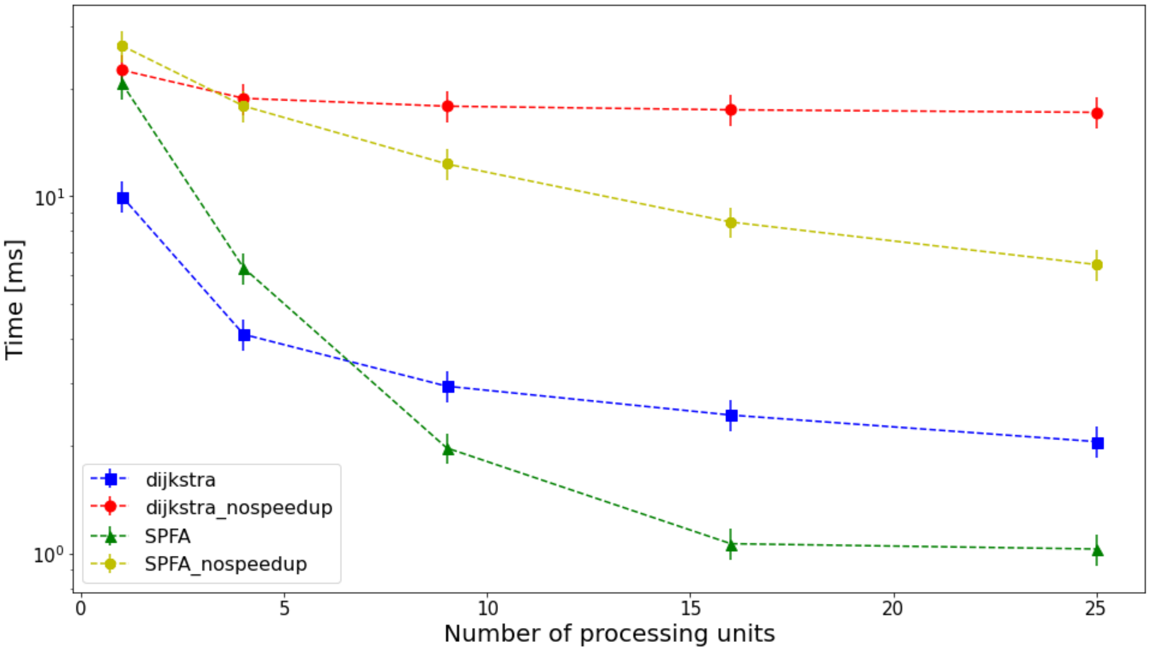
The optimization proposed decreases the communication cost which then improves on the total work performed by both algorithms. The size of the effect varies depending on size of the graph processed. Execution time of SPFA is also improved, but for Dijkstra's algorithm the results are most likely due to random noise in the data.

4.6 Comparison of single–source algorithms

Dijkstra's algorithm and SPFA can process much larger graphs quickly if we do not require calculations of shortest paths between all pairs of vertices. Thus, for a complete comparison we need to also check if large graphs reverse the trends we've seen previously. I have downloaded the entire road networks of Cambridge, Szczecin[3], Rome and London. I have picked 5 random locations and used the SSSP algorithms to calculate distances from those locations, then averaged the results to report a score for a particular graph.

The outcome of this investigation proved that Dijkstra's algorithm is a highly efficient algorithm when only a single core is available, as for the two largest networks it outperformed SPFA. Nonetheless, as P increases, SPFA can utilize the processing elements better.

Figure (4.2) Averaged execution times for the London network. $N = 126917$, $M = 297832$. Nospeedup versions use random assignment of vertices to workers.

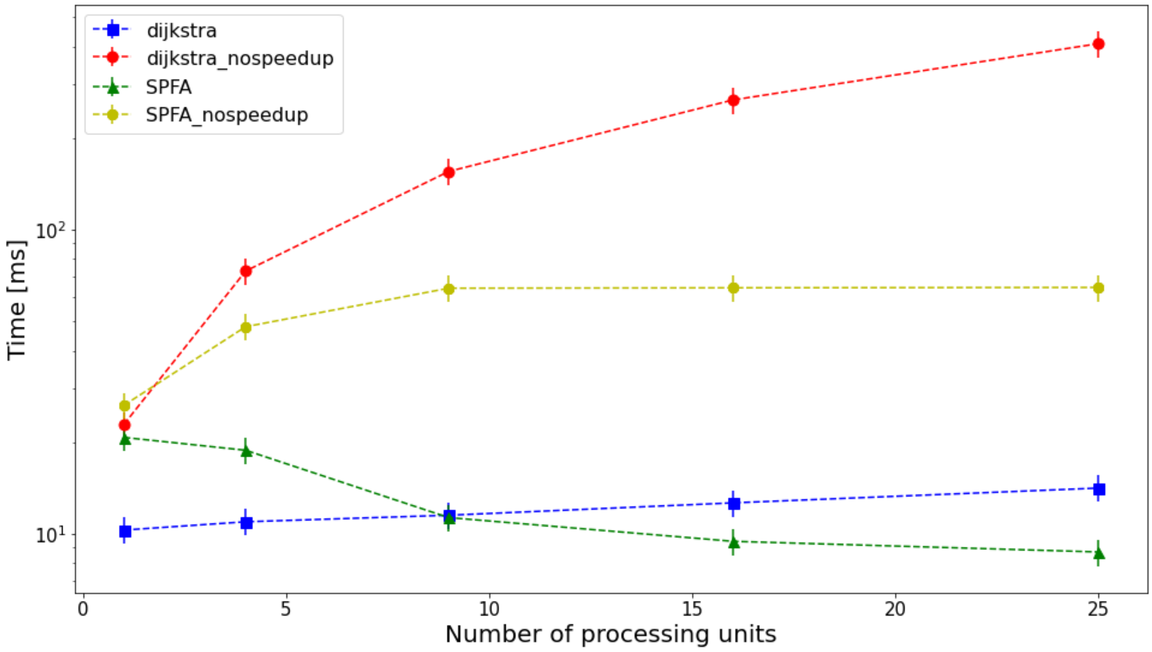


Additionally, the single–source experiments verify that optimizing the communication costs is crucial for large problem instances. Versions of the algorithms without this feature lose beyond doubt.

The measurements for total work performed are a bit confusing. As more cores are involved, SPFA performs less and less work! The phenomenon repeatedly arises on the largest test instances. One explanation which I came up with was an insidious bug in the simulator which only manifests on very large graphs. I don't see a convincing reason why such a bug would manifest only on large graphs. The question remains open.

[3] my home city, roughly $400K$ inhabitants

Figure (4.3) Total work performed for the London network. $N = 126917$, $M = 297832$



4.7 Sensitivity analysis

Although the constants described in section 4.2 were chosen to reflect a possible real–life scenario, an inquisitive reader may wonder what would happen if they were different and how important the choices are. In this section I am describing plausible answers to these questions.

I've started the analysis by measuring means and standard deviations of the reported results over 10 independent runs on the same test instance ($N = 154$), and calculated the relative standard deviations. These coefficients did not exceed 0.0015, as the analysis in section 4.2 predicted.

What would happen if I used different constant for the parameters? To answer this question, I needed to determine the impact certain operations have on the final measurements. Initially I wanted to create a matrix of partial derivatives whose entries calculate how much a given result changes when one of my constants is perturbed by a small amount. Unfortunately, it would be computationally expensive to produce a detailed result of this kind, so I needed to seek for a good approximation. The standard deviations were proven to be small, so the total measurement could be reasonably approximated with a linear function. This meant that I could calculate the contributions as fractions of the final result.

Let's denote C_i as the cumulative costs of the i–th operation, for example of all performed memory reads, and T_i as the total cost but **without** taking the i–th operation into account. If the unit cost of a memory read becomes x times larger, the contribution of memory reads becomes $\frac{C_i \cdot x}{T_i + C_i \cdot x}$ instead of $\frac{C_i}{T_i + C_i}$, which can be calculated easily to answer any hypothetical questions.

The following plots describe the breakdown of operation costs in the total work performed on aforementioned graph with $N = 154$ vertices when $P = 4$ and $P = 16$ cores, averaged over 10 runs. The algorithms differ in runtime significantly, therefore the normalised plots are also presented. Although the time cores need to wait for the messages is not related to any explicit constants, I felt that it is important to measure it, too.

Figure (4.4) Breakdown of operation costs

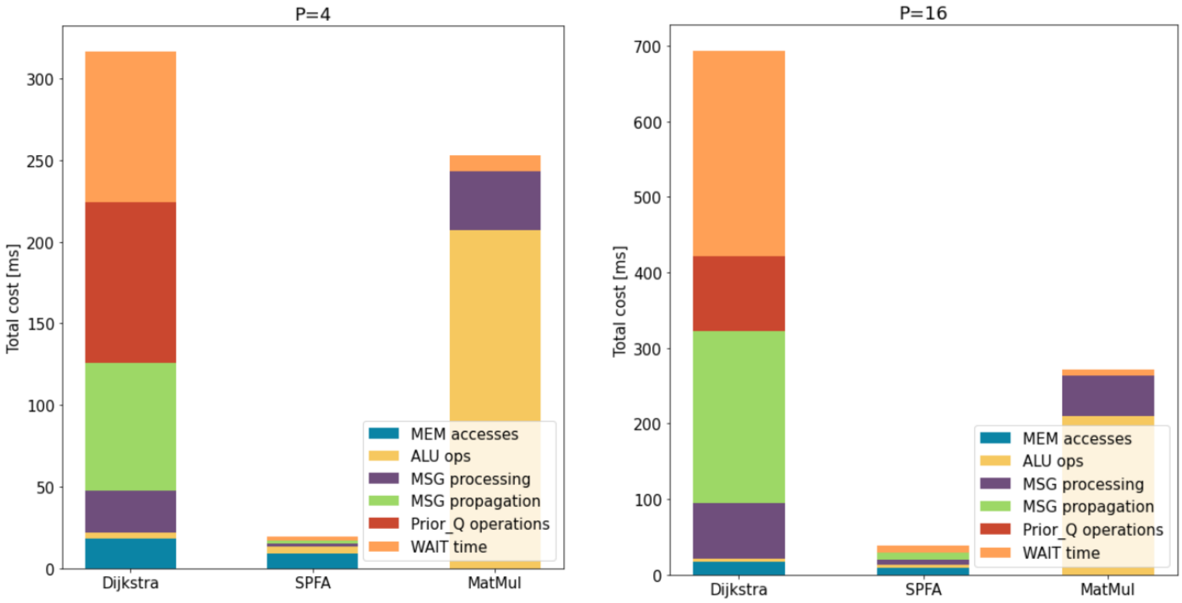
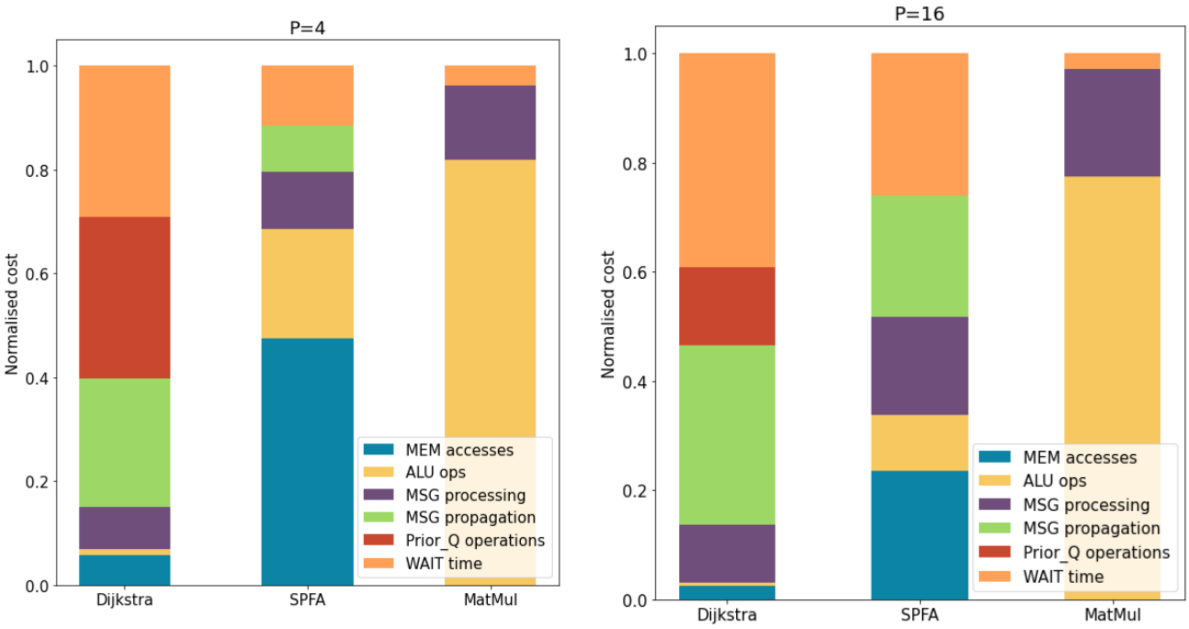


Figure (4.5) Normalised breakdown of operation costs



A couple of interesting remarks can be made for each of the algorithms:

- Dijkstra's algorithm performs a lot of work overall. The key factors which contribute to the poor results are delays arising from synchronization needs: time spent waiting and additional messages sent in order to cooperate. Additionally, although priority queue overheads are significant, the algorithm would still perform poorly if they were negligible.

- SPFA has a tiny fraction of Dijkstra's runtime, and spends majority of time doing useful work. However, as P increases, the costs associated with message passing rise sharply.

- MatMul's cost is dominated by ALU operations, as the time for reading matrices is also incorporated there. Message propagation delay is near 0, but message processing is not – this is because there are relatively large matrices being passed in very few messages. Wait time is close to zero too, as the processing elements execute almost identical instructions.

Chapter 5

Conclusions

How much algorithm redesign is needed for parallel architectures? To phrase it concisely: a lot. In my project I have studied various approaches to the shortest path problem to compare their relative merits. My results show that the most appropriate algorithm may depend on many factors: the number of processing elements available, the size of the problem instance, the costs of communication between workers, and more. For example, a matrix multiplication algorithm, inefficient in sequential environment, may be the best choice if there is a very large number of processing elements available.

In my work I have considered the SPFA algorithm which, to my knowledge, has never been evaluated in the context of parallel algorithms before. The results surpassed my expectations: Parallel SPFA is a fast and work–efficient parallel algorithm which consistently outperforms the parallel adaptation of its sequential cousin - Dijkstra's algorithm. Having analysed the two, I have concluded that algorithms which require strong sequential consistency requirements are inappropriate for parallel environments - maintaining those is a waste of time and energy.

5.1 Success criteria

All the success criteria of my project have been met. I have implemented a test bench for parallel algorithms and tested three different parallel algorithms using this framework. Graphs acquired using the OSMnx library were used to evaluate the implementations in terms of execution time and total work performed. Although the success criteria did not promise anything in terms of performance, I am proud to conclude that my parallelization of the Shortest Path Faster Algorithm led to the most efficient algorithm overall.

5.2 Reflections

This project made me realize how incredibly rich the field of parallel algorithms is. They are far more challenging and relevant to modern computing than the traditional sequential algorithms. I have underestimated the time needed to design and implement good, modular software – matching the simulator milestones on schedule proved to be impossible. If I had to redo the project, I would also include *'Evaluation using cloud facilities'* among the success criteria, to save myself waiting many hours for the evaluation results and to enable more comprehensive evaluation on larger graphs.

5.3 Future work

The area of parallel algorithms is a broad field which cannot be explored in too much detail during a Part II Project. In my opinion, the most exciting future work would build on the success of SPFA in this project and try to apply similar chaotic relaxation techniques in the field of automatic parallelization of programs. A study on the usefulness of such transformations could be of great interest to general community.

Bibliography

[1] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: https://doi.org/10.1145/324133.324234.

[2] Geoff Boeing. “OSMnx: A Python package to work with graph-theoretic OpenStreetMap street networks”. In: *Journal of Open Source Software* 2.12 (2017), p. 215. DOI: 10.21105/joss.00215. URL: https://doi.org/10.21105/joss.00215.

[3] Lynn Elliot Cannon. “A Cellular Computer to Implement the Kalman Filter Algorithm”. AAI7010025. PhD thesis. USA, 1969.

[4] “Central Limit Theorem”. In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 66–68. ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1_50. URL: https://doi.org/10.1007/978-0-387-32833-1_50.

[5] Andreas Crauser et al. “A Parallelization of Dijkstra’s Shortest Path Algorithm”. In: *Proceedings of the 23rd International Symposium on the Mathematical Foundations of Computer Science (MFCS-98), Springer, 722-731 (1998)* (Oct. 1998). DOI: 10.1007/BFb0055823.

[6] L. Eeckhout. “Is Moore’s Law Slowing Down? What’s Next?” In: *IEEE Micro* 37.04 (July 2017), pp. 4–5. ISSN: 1937-4143. DOI: 10.1109/MM.2017.3211123.

[7] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[8] Rodney Howell. “On Asymptotic Notation with Multiple Variables”. In: (Jan. 2007).

[9] Feifei Li et al. “On Trip Planning Queries in Spatial Databases”. In: vol. 3633. Aug. 2005, pp. 273–290. ISBN: 978-3-540-28127-6. DOI: 10.1007/11535331_16.

[10] Jagdish Modi. “Parallel algorithms and matrix computation”. In: *Oxford University Press* (1988).

[11] Prema Soundararajan et al. “Dependency Analysis and Loop Transformation Characteristics of Auto-Parallelizers”. In: Feb. 2015. DOI: 10.1109/PARCOMPTECH.2015.7084524.

[12] Nadya Travinin et al. “Automatic Parallelization with pMapper”. In: (Oct. 2005), pp. 1–2. DOI: 10.1109/CLUSTR.2005.347017.

[13] D.H. Wolpert and W.G. Macready. “No free lunch theorems for optimization”. In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82. DOI: 10.1109/4235.585893.

Appendix A

Detailed repository overview diagram

```
route-planning
├── data
└── src
    ├── algorithms
    │   ├── APSP.MatMul
    │   │   ├── CannonCore.java
    │   │   ├── CannonMatMul.java
    │   │   ├── MatMulAlgorithm.java
    │   │   └── MatMulPolicy.java
    │   ├── SSSP
    │   ├── Dijkstra
    │   │   ├── Dijkstra.java
    │   │   ├── DijkstraCore.java
    │   │   └── DijkstraNoSpeedup.java
    │   ├── SPFA
    │   │   ├── SPFA.java
    │   │   ├── SPFACore.java
    │   │   └── SPFANoSpeedup.java
    │   ├── SSAlgorithm.java
    │   └── SSPolicy.java
    └── graphs
        ├── cache
        │   ├── Assignment.java
        │   ├── Edge.java
        │   ├── evaluation.ipynb
        │   ├── Graph.java
        │   ├── graph_download.ipynb
        │   └── Matrix.java
        └── simulator
            └── internal
                ├── CommunicationHandler.java
                ├── ConcurrentQueue.java
                └── ConcurrentQueueImpl.java
```

```
│  │  │
│  │  ├─Scheduler.java
│  │  ├─SingleConnectionHandler.java
│  │  └─TaskRunner.java
│  └─utils
│     ├─Core.java
│     ├─Estimator.java
│     ├─EvaluationEnvironment.java
│     ├─FibonacciHeap.java
│     ├─GlobalArray.java
│     ├─GlobalBitset.java
│     ├─GlobalQueue.java
│     ├─Lattice.java
│     ├─LocalArray.java
│     ├─LocalHeap.java
│     ├─MasterCore.java
│     ├─Message.java
│     ├─Metrics.java
│     ├─OpTracker.java
│     ├─Phase.java
│     ├─ProcessorArchitecture.java
│     └─Task.java
└─test
   ├─DijkstraMultiThreadedTests.java
   ├─DijkstraNoSpeedupMultiThreadedTests.java
   ├─DijkstraNoSpeedupSingleThreadedTests.java
   ├─Evaluation.java
   ├─MatMulAlgorithmMultiThreadedTests.java
   ├─MatMulAlgorithmSingleThreadedTests.java
   ├─SPFAMultiThreadedTests.java
   └─SPFANoSpeedupMultiThreadedTests.java
```

Directory `data/` contains data generated automatically generated with help of OSMnx, but it would be pointless to include those.

Appendix B

Plots comparing different versions of SPFA and Dijkstra

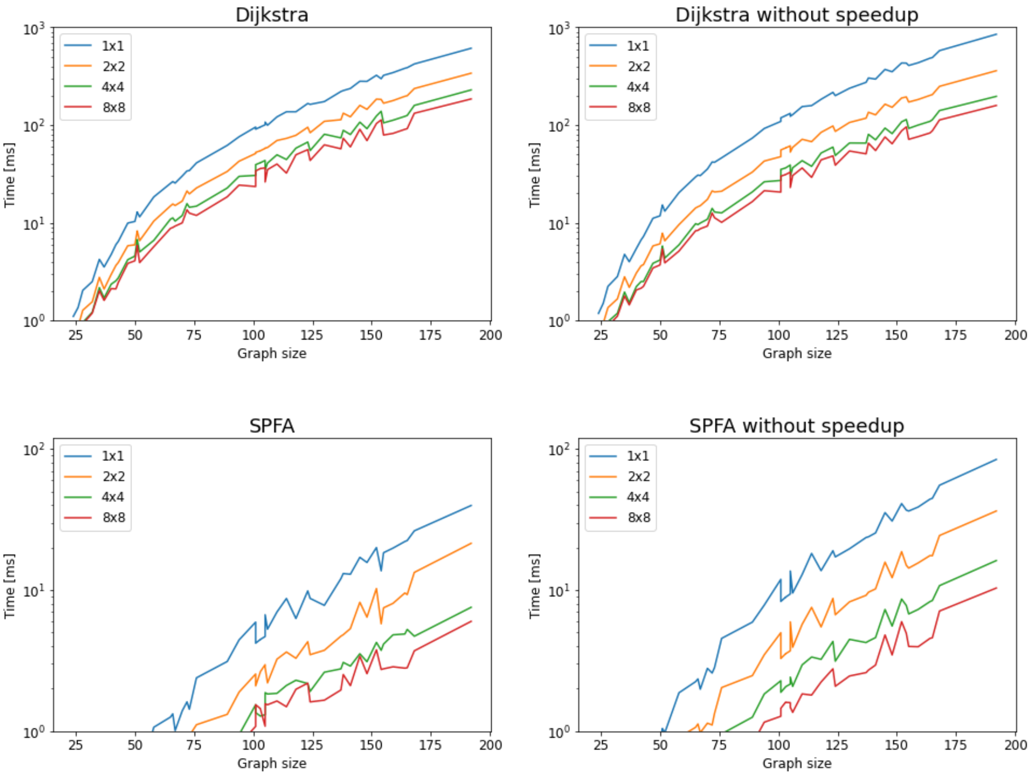


Figure (B.1) Comparison of runtime among the algorithms and their non–speedup versions

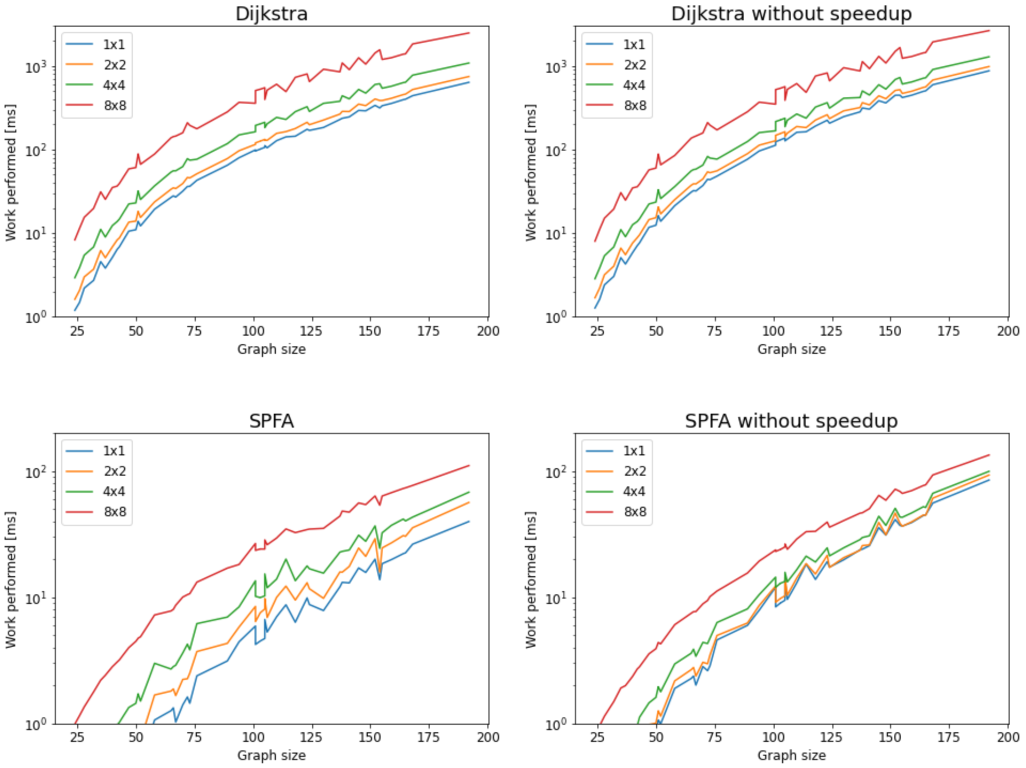


Figure (B.2) Comparison of total work performed among the algorithms and their non–speedup versions