Marcus A. K. September

A parallel algorithm for all-pairs shortest paths that minimises data movement

Computer Science Tripos – Part II

Clare College

May 3, 2022

Declaration

I, Marcus A. K. September of Clare 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.

Signed

Date: May 3, 2022

Acknowledgements

TODO

Proforma

Candidate Number:	**2406F**
Project Title:	**A parallel algorithm for all-pairs shortest paths that minimises data movement**
Examination:	**Computer Science Tripos – Part II, May 2021**
Word Count:	**11134**[1]
Final line count :	**4081 (Java, `main/`) + 1796 (Java, `test/`) + 728 (Python)**
Project Originator:	Dr J. Modi
Supervisor:	Dr J. Modi

Original Aims of the Project

The aim was to implement a massively parallel algorithm based on matrix-multiplication for solving all-pairs shortest paths (APSP). For this, a parallel system were to be simulated using Java. I would then evaluate the advantage of this parallel algorithm for solving APSP, and investigate how this advantage changed with increasing problem sizes.

Work Completed

The project was successful. All the success criteria were met, and three extensions were completed. I implemented a an expressive simulator of a massively parallel system with an interface that was simple to use. A parallel APSP algorithm based on matrix multiplication was then implemented using this interface, and was further generalised. I then extended the simulator with sophisticated functionality for timing execution, and optimised the algorithm with graph compression. Finally, I did a qualitative analysis on the benefit of parallel computation where I considered different types of systems and problem sizes.

[1] This word count was computed using TEXcount. See appendix B.1 for more details.

Special Difficulties

None

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms and abbreviations

APSP *All-Pairs Shortest Paths*

SSSP *Single-source shortest paths*

MatSquare APSP via repeated matrix-squaring

SISD Single-instruction-single-data

MISD Multiple-instruction-single-data

SIMD Single-instruction-multiple-data

MIMD Multiple-instruction-multiple-data

GPU Graphics processing unit

CPU Central processing unit

PE Processing element

MPP Massively parallel processor

OOP Object-Oriented Programming

JVM Java virtual machine

Chapter 1

Introduction

The dissertation proposes using an algorithm based on matrix multiplication for solving *All-Pairs Shortest Paths* (APSP), and demonstrates that parallel computation has significant benefit for solving APSP on real-world graphs. With this aim, I simulate a parallel system with characteristics based on real hardware. The simulator provides an expressive interface for developing parallel applications, which is used to implement the APSP algorithm. I then evaluate the algorithm through execution-time measures and other tests. The overall project has been a great success as both the base success criteria and several extensions were achieved.

1.1 Motivation & aims

1.1.1 Parallel computation

Since the 1970s, there has been a yearly exponential increase in the number of transistors in integrated circuits, thanks to Moore's law. However, around 2006, Dennard scaling started to break down, which meant that the increase in transistor density did not directly translate to an increase in performance per watt on a single processor As a result, performance improvements had to be achieved through other means. The innovation of multi-core processors allowed the aggregated performance per watt to keep increasing by combing many processing elements (processing elements (PEs)). Today, there are various different parallel systems.

Designing algorithms to efficiently utilise the computational resources of parallel systems is important. There is a limit to how much automatic parallelisation can increase performance of a serial algorithm. Therefore, the performance of such algorithms will not scale well with the current trends in processor manufacturing. Instead, we should go back to the problem to be solved and rethink an algorithm with parallel computation in mind. That way, we can utilise the available PEs fully and increase performance to a high degree.

1.1.2 The all-pairs shortest path problem

In the shortest path problem, the goal is finding the *shortest path* in a graph $G = (V, E, w)$ from vertex i to vertex j. In the APSP variant of the problem, we want to find the shortest paths between all pairs of vertices $(i, j) \in V^2$. By working with weighted graphs, a path-finding algorithm can be applied in many different areas. For example, it can be used for planning routes in transport networks, routing packets across the Internet according to some metric, or to plan a robot's movement.

1.1.3 Related work and proposed analysis

There is a plethora of different parallel algorithms to solve route-planning problems [6, 3, 14, 11, 10]. However, the evaluation of the proposed or surveyed algorithms differ greatly from paper to paper. In some research, the analysis of how the execution time scales with the problem is mostly based on the algorithm's asymptotic complexity [11, 3]. Kim et al. and Sao et al. have analysed the performance of their algorithms through experiments, but they did not run them on more than 256 PEs. Additionally, their test graphs are dense, randomly generated graphs, not based on real-world graphs such as road-networks.

This project aims to analyse the scaling of a parallel algorithm for solving APSP on real-world-like graphs, using a simulation of a parallel system with a massive number of PEs. The project will also investigate the benefit of parallel execution for different problem sizes and parallel systems, considering both the number of PEs and the class of the parallel system.

1.2 Project structure

In the Preparation chapter, I give an overview on the different types of parallel systems as well as listing the assumptions I made for the system to be simulated. The concept behind the APSP algorithm is briefly explained as well. Afterwards, I present an analysis of the requirements of each component to be implemented, and cover the tools and engineering techniques I will use during the implementation phase.

In the Implementation chapter, I cover the graph datasets used. Then I explain the parallel system simulation in a top-down fashion, emphasizing the interface it provides to the programmer. I then move onto explaining the algorithm for solving APSP, how it is parallelised, and how it is further optimised through compression of the input graph. Before any code was written, I studied a lot of theory behind this algorithm, but this is presented together with its implementation for increased readability.

In the evaluation chapter, I relate implemented components to corresponding success criteria. I also reference several tests used to justify the correctness of the software produced. Afterwards,

I present plots of the runtime measurements and use these to discuss the advantage of parallel computation for solving APSP.

Chapter 2

Preparation

This chapter gives an overview of work that was completed before any code was written. This includes theory surrounding parallel computing (section 2.1) and computing shortest paths through matrix multiplication (section 2.2). It also includes the planning the components of the implementation, done through making up assumptions explicit (section 2.3) and outline the requirements (section 2.4). Lastly, I go over the tools used (section 2.5) and the techniques used for productive and manageable development (section 2.7).

2.1 Parallel computing

We will now look at an overview of parallel computers. To design parallel algorithms that utilize the computational resources as efficiently as possible, we should be familiar with the landscape of parallel systems and their classifications. In this section, I will provide such an overview by classifying systems according to how the streams of instructions are mapped onto data, what hierarchies of memory are commonly used, and how this affects inter-processing-element communication. Additionally, we will discuss the scale at which parallelisation happens and cover metrics used for evaluating how resource efficient a parallel algorithm is.

2.1.1 Flynn's taxonomy

One way to classify parallel systems, is to look at how many streams of data there are and how many sets of instructions the computer is operating with.

- Single-instruction-single-data (SISD) – This is the same as sequential execution.
- Multiple-instruction-single-data (MISD) – Here, multiple sets of instructions are executed on the same input data. There are very few systems using this type of parallelism.
- Single-instruction-multiple-data (SIMD) – With this scheme, we apply the same set of operations to multiple sets of data simultaneously.

- Multiple-instruction-multiple-data (MIMD) – This is the most common type of parallelism in hardware. Multiple sets of instructions, possibly different, are applied to multiple sets of data at the same time.

Parallel systems typically use either SIMD or MIMD, where SIMD is common in graphics processing units (GPUs) and MIMD is more often seen in systems where the computation is spread among central processing units (CPUs).

2.1.2 Memory and communication

We can also classify parallel systems by how the memory is laid out. In systems that use *distributed memory*, each processor has its own private memory. If communication between processors is required, there will be an interconnection between the PEs that allows message passing. One common approach is an interconnect network that can be arranged in a plethora of different *topologies*. If the topology is not fully-connected, the messages need to be routed. A common network arrangement is the 2D Lattice Mesh, where there are p^2 PEs connected to each of their four orthogonal neighbours and there are cyclic connection that wrap around the edges. In *shared memory* systems, all the processors have access to a shared address space. Multiple PEs can communicate by reading and writing to the same segment of memory, and locking mechanisms can be implemented to prevent race conditions.

2.1.3 Classes of parallel computers

By looking at the extent to which the system supports parallelism and at the physical distance between the PEs, we create further classifications. In *multi-core processors*, we refer to the PEs as *cores* and have several of them on the same chip. The cores execute in MIMD-fashion and there are typically both local memory for each core as well as some shared memory between cores, that can be used for communication. In massively parallel processors (MPPs), we often have many thousands of processors, interconnected through a specialised network. An example is the Sunway TaihuLight supercomputer [4]. There is also a wide class of computers known as *distributed computers*, where the PEs communicate over a network, such as the Internet.

2.1.4 Evaluating performance

A simple performance metric is the elapsed time, which works well when comparing two different computers. However, it is not very useful when evaluating how efficiently an algorithm exploits the parallelism available on a given system. For this, the *speed-up* is more useful:

$$S_p = \frac{T_1}{T_p}, \tag{2.1}$$

where T_1 is the required time to solve the problem on a single processor and T_p is the elapsed time when executing the parallel algorithm on p processors.

Another useful metric is the *parallel efficiency*, which is defined as

$$\varepsilon = \frac{T_1}{pT_p} = \frac{S_p}{p}. \tag{2.2}$$

The parallel efficiency ranges from 0 to 1, where $\varepsilon = 1$, means we have a perfect linear speed-up where no computation power is wasted when executing the algorithm on our parallel system.

Both of these metrics do not take into account the speed-up lost from the algorithm having a serial part that cannot be parallelised. To incorporate this, Amdahl's law or Gustafson's law have been used [9]. However, the parallel algorithms developed in this project to solve APSP have a negligible serial part if we ignore the time required to read the input and print the result. I will therefore not use these laws in evaluation.

In practice, what prevents linear speed-up is the *communication cost* the algorithm incurs while running on the parallel system. This can include both PEs needing to idle while they wait for data to be ready or they are waiting for some data to be transferred. Another metric is therefore the ratio between computation and communication, where a high ratio means we use our resources efficiently. We let $T_p = \frac{T_{communication}+T_{computation}}{p}$, where T_p is the time spent on executing the parallel algorithm, where the sum of the costs are split up into communication and computation costs. To execute a similar algorithm on a serial machine, we would avoid all the communication cost, but would still need to do all the computation, so $T_1 = T_{computation}$[1]. We can relate this with the parallel efficiency:

$$\varepsilon = \frac{T_1}{pT_p} = \frac{T_{computation}}{T_{computation} + T_{communication}} \tag{2.3}$$

It is also easier to measure this ratio of computation.

2.2 APSP algorithm

The main algorithm of the dissertation is based on matrix multiplication and solves a problem in graph theory. By going through the notation and definitions I will be using, as well as discussing alternative approaches to parallelising APSP, we can better understand the APSP via repeated matrix-squaring (MatSquare) algorithm. Throughout the dissertation, we consider graphs on the form $G = (V, E, w)$, where we have a set V of n vertices and a set E of edges and each edge $e \in E$ has a non-negative weight $w(e)$ assigned to it. The APSP problem concerns finding the *shortest path* between each pair of vertices.

[1] This is assuming we are not using a completely different algorithm when solving the problem serially.

2.2.1 Parallelisation

There are many widely-used algorithms for efficiently solving the *Single-source shortest paths* (SSSP) problem, such as Dijkstra's algorithm, Bellman-Ford and A*-search. If we have fewer than n PEs, a trivial parallelisation for solving APSP is to run one of these algorithms on each PE, distributing the sources. However, this approach does not use our parallel resources efficiently if we have more than n PEs. To exploit this, we must either consider other algorithms or parallelise parts of the SSSP computation in one of these algorithms. Parallel versions of these algorithms have been developed in the past [3, 6], but they incur a significant overhead. I have therefore decided to use a new algorithm based on matrix multiplication, a highly paralleliseable task, instead.

2.2.2 Repeated matrix squaring

Matrix multiplication usually happens over the *semiring*[2] $(\mathbb{R}, +, \cdot)$. However, we replace addition with multiplication and multiplication by the min-operator, resulting in the tropical semiring $(\mathbb{R}^*, \cdot, \min)$, where we define $\mathbb{R}^* \triangleq \mathbb{R} \cup \{\infty\}$. When doing matrix multiplication over this algebra, we perform an operation called the *distance product*.

Definition 1. *The distance product of two matrices, A and B, whose entries (i, j) give the length of the shortest path $i \rightsquigarrow j$ of length exactly n and m, respectively, is a matrix M whose entries (i, j) are given by*

$$M_{i,j} = [A \otimes B]_{i,j} = \min_k \left(A_{i,k} \cdot B_{k,j}\right). \tag{2.4}$$

Each entry $M_{i,j}$ then gives the distance of the shortest path $i \rightsquigarrow k \rightsquigarrow j$ of length $n + m$.

If we let W be the adjacency matrix, where each entry (i, j) is the weight of edge (i, j) and ∞ if $(i, j) \notin E$, then computing the distance product of W with itself k times will give a matrix containing the length of shortest paths of length k. If we add a self-loop to each vertex with weight 0, and compute W^n, the matrix will contain the shortest-distances between all pairs of nodes that are of length at most n, i.e. the shortest path (because it cannot be longer). A more efficient way to compute this is by squaring W $\lceil \log_2 n \rceil$ times. This is the idea behind the MatSquare algorithm that I explain in section 3.3.

2.3 Parallel architecture assumptions

In this section, I will list the assumptions behind the parallel system that will be simulated. For the implementation to go as smoothly as possible and the timing evaluation to be applicable, it is important to have a clear and well-defined parallel system in mind. As we saw in section 2.1,

[2] See appendix section A.1.

there are many classes of parallel systems. We will hone in on a particular parallel system by making assumptions about its model of memory, its execution, and how it supports message-passing.

Memory model I assume a fully-distributed memory model, where each PE has its own private memory and no address space is shared between different PEs. To allow coordination, a message-passing interface is also assumed. This model scales well with a very large number of PEs, compared with shared-memory systems where the memory system becomes a bottleneck when the number of PEs that share the same memory are in the 1000s. I also assume that each PE has sufficient private memory to store the data it will be working on, but I will not abuse this to store a copy of the input on each PE.

MIMD execution I also assume that each PE runs independently of the others, so their computation might not be synchronised even if they all execute the same algorithm. I have decided to assume this model of execution over SIMD for the following reasons:

- In distributed-memory systems with message passing, MIMD is the most common, whereas SIMD is most-often seen in shared-memory systems. Simulating MIMD thus gives more realism.
- With SIMD, if two or more PEs take different branches, computation is masked with `NOP`-instructions. This wastes computation resources, and is difficult to simulate. In standard matrix multiplication, there are no branches, but we are implementing a alternative matrix multiplication algorithm for the $(\mathbb{R}^*, \cdot, \min)$ semiring, where we also have computation for finding the predecessors. This introduces possibly large branches in our inner-loop.

In the initial plan for the project, I set out to use a naïve simulation of SIMD, where I would just average the computation time of each PE. The change to a more realistic MIMD execution model was an extension that was not planned for in the project proposal.

Message passing The PEs are interconnected through point-to-point links, arranged in some connected network topology. I will assume the PEs are laid out in a square lattice arrangement because the problem of matrix multiplication maps nicely to this. Additionally, the Fox-Otto algorithm assumes row-broadcast highways, so I will also assume these are available. Further, I will assume that each broadcast-highway as well as each point-to-point link can only be used to carry one message at a time. The messages are also delivered *in-order*.

The time it takes to send a message m from PE i to PE j only depends on the size of m, the latency and bandwidth of the links, and the value of some fixed function $d(i, j)$, that depends on the interconnect topology. I will also assume the latency and bandwidth are the same for all messages sent. I also assume some fixed latency and bandwidth for the broadcasting.

2.4 Requirements analysis

We will now look at the main components to be implemented, and what their functional requirements are. To make implementation as smooth as possible, it is important to know beforehand what the different components will do and how they interface with other components. This will also make modular development easier as we only need to focus on one component and its interface at a time.

Parallel system simulation Since I do not have access to a physical system fitting the properties described in section 2.3, I will simulate one. The main requirements of this is:

- Configurable number of PEs and topology
- Simulation exploits inherent parallelism on system it is executed on, such that it's as efficient as possible
- There is a simple, yet expressive, interface for describing the computation to be performed at each PE, where the programmer has access to methods such as `send`, `receive`, `broadcastRow` etc., and can retrieve its position in the grid with `i` and `j`. With such an interface, other algorithms like parallel Floyd-Warshall can also be implemented.
- There is an interface for distributing input data among the PEs as well as retrieving the result.
- The computation time is measured during PE execution, and communication time is estimated according to the message passing model in the assumed architecture.

Graph datasets To evaluate the performance scaling of our algorithms, we will need a large set of different input graphs, covering a sufficiently large space of sizes. Ideally, these graphs should have similar characteristics as well as resemble real-world graphs like road networks as this is a primary usage of our APSP algorithm.

We also need a *graph input* component that can read the graphs from file and transform them into a suitable format to pass as input to the APSP algorithm.

APSP algorithm We can summarise the requirements of this component as:

- Given an input graph G, it can compute the shortest paths between each pair of nodes in V. This computation is done by using the interface for the parallel architecture simulation.
- After this computation, each path $i \rightsquigarrow j$ can be reconstructed.
- This computation can also happen when the parallel arcitecture is configured to have fewer PEs than there are vertices in the graph (extension).

Graph compressor (extension) The input graph can be passed to this module and a new "equivalent", but with all the 2-degree-nodes removed is returned. Additionally, after solving APSP on this compressed graph, we can pass the solution back to the graph compressor to reconstruct APSP solutions on the original graph.

2.5 Choice of tools

We will now go over various existing software, programming languages and services that can aid the development. When developing and testing a parallel program, there are many already built tools and utilities that handle a lot of the work, such as high-level synchronisation classes or unit testing frameworks. By using these tools, we can spend more time on the core parts of the project, like creating the simulation and developing the MatSquare algorithm.

2.5.1 Programming languages and libraries

I chose Java for the implementation of the components of the project. The parallel simulation has many modules that interact with each other, and implementation is more manageable if these can be developed in isolation, abstracting away their functionality. Additionally, code-reuse and dynamic polymorphism will be important when implementing different algorithms for the parallel system's interface. As such, and object-oriented language like Java, which makes following these principles easier, is suitable. Additionally, Java is well-suited for creating concurrent programs as it has built-in primitives for synchronisation as well as classes for managing a large number of concurrent threads.

For testing, I use JUnit[3], which is a testing framework for the Java virtual machine (JVM). Since the code is modularised, this will allow me to create well-organised unit tests for the different modules. To generate the random graphs, I use the `NetworkX` library in Python, and these graphs can also be visualised using a `ForceAtlas2` library. For plotting my evaluation results, I use python and modules like `pandas` and `matplotlib`.

2.5.2 Development, testing and revision control

To easily manage different versions of the codebase, I will do revision control with `git` and GitHub[4]. I also set up contiguous integration with GitHub Workerflows, which compiles the project every time I push a commit. This way, I can make sure the code is always in a valid state and there are no syntax errors. I will use my Acer Nitro Laptop, running Ubuntu Linux 18.04 LTS, to both write and run all of my code. It has a quad-core CPU (an Intel i5-8300H

[3] `https://junit.org`
[4] `https://github.com/`

2.30 Ghz), 8 GB of RAM, and an NVIDIA GTX 1050 Mobile. The CPU supports running 8 threads concurrently, which means parallelising the simulation itself will make a significant impact on the simulation time. To write the Java code, I use the IntelliJ IDEA IDE to improve the quality of the code and spend less time on getting the Java syntax right.

2.6 Starting point

Course	**Relevance**
Object-Oriented Programming (OOP)	Principles behind writing OOP code and managing a large codebase
Further Java	Some experience writing parallel programs
Concurrent and Distributed Systems	Knowledge of concurrency concepts
Computer Design	Knowledge of parallel systems
Algorithms	Serial route-planning algorithms

Table 2.1: Courses from the Tripos and their relevance to my project.

Many courses from the Computer Science Tripos have been useful for my project, as seen in table 2.1. The unit of assessment, Advanced Data Science, has also been applicable when managing and visualising the timing data gathered for the evaluation.

2.7 Software engineering

In Figure 2.1, we see the high-level structure of the implementation. This matches the requirements laid out in section 2.4; The requirements in each group can also be associated with individual components. For example, allowing descriptions of computation comes from the `Worker` interface, while access to communication methods come from implementation of the communication manager. This sets up clear work items and achievements to aim for during implementation.

I developed the parallel system simulation using the *incremental build model*. The simulator is very modular and the component dependencies form a directed acyclic graph, so implementation could happen by developing one module at a time, and incrementally expanding the simulator. I started implementation with the `Worker` and its private memory, and tested its independent execution using various test-sub-classes. I then moved onto incorporating communication methods such as `send` and `broadcastRow` with the communication manager, which I tested using more sub-classes of the `Worker` interface. After this, managing several workers at a time, and handling their automatic creation, was implemented with the `Manager` class and the worker factory, respectively. Then I incorporated the communication manager. The functionality for timing the execution is implemented using the *decorator* pattern, so this could incrementally be developed at any time after the parallel system simulation was done.

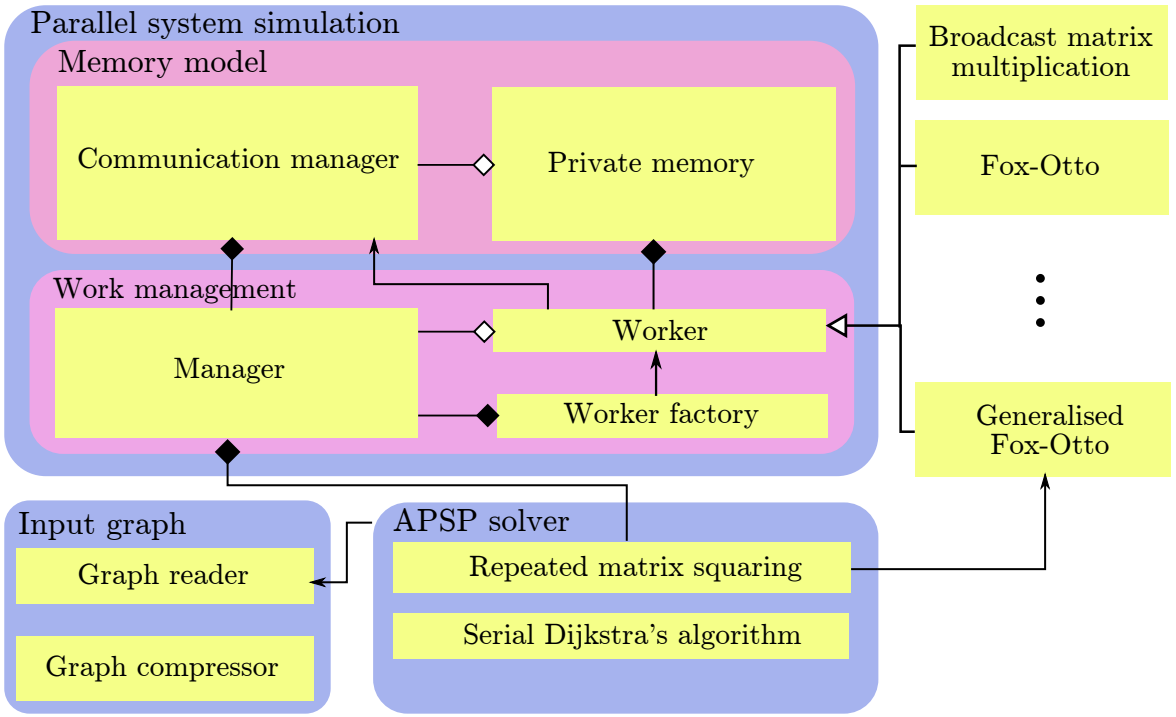


Figure 2.1: High-level overview of the implementation

The higher-level components such as the simulation, the APSP solver and the input graph module were implemented according to the *iterative development model*. These components were fairly independent as we could use a serial matrix multiplication algorithm instead of running code on the parallel system. This allowed me to focus on one component at a time, as well as separate testing.

I followed proper software engineering techniques during development. Encapsulation and modularisation were widely used and realised through use of immutability, access control mechanisms and packaging my code. I also wrote exceptions and assertions throughout my code where it was appropriate. I also used a coherent coding style and documented both my methods and classes according to the Javadoc[5] standard. The various components were also unit-tested.

2.8 Summary

In this chapter, I reviewed work done before code was written. I presented a brief overview of different classes of parallel systems and the concept behind the APSP algorithm. I also laid out the requirements for the different parts of the project, emphasising the assumptions made about the parallel system to be simulated. Then I outlined the choice of tools and techniques to be used during development.

[5] `https://www.oracle.com/java/technologies/javase/javadoc-tool.html`

Chapter 3

Implementation

This chapter elaborates on how the final codebase, as shown in figure 3.1, accomplishes all the requirements laid out in section 2.4. I start by covering the input graphs in section 3.1. Afterwards, I explain how I simulated a parallel system in section 3.2. The simulator provides an expressive, yet simple, interface for writing and timing the execution of massively parallel programs. In section 3.3, this interface is used to implement the MatSquare algorithm, which is a massively parallel algorithm for solving APSP. Finally, I describe an optimisation to the algorithm in section 3.4.

3.1 Graph datasets

To run the algorithms, we need graphs as input. Since one important usage of APSP is route-planning, I used a dataset of the Californian road-network. This dataset was initially presented by Li et al. and has been made available on the author's website[1] [12]. This graph has 21048 vertices and will be used to prove that the MatSquare algorithm can efficiently solve practical route-planning problems on a parallel system. However, to evaluate the algorithm's scalability, I need graphs of a wide range of different sizes. These were generated randomly.

3.1.1 Random graph generation

To generate the graphs, I used Erdös-Rényi graphs, specifically the $G(n, p)$ model: We start with an edge-less graph of n vertices. We then independently include each of the n^2 edges with probability p. I chose this model because it allows specifying the graphs size, and it has previously been used to evaluate the performance of APSP algorithms [3].

[1] `https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm`

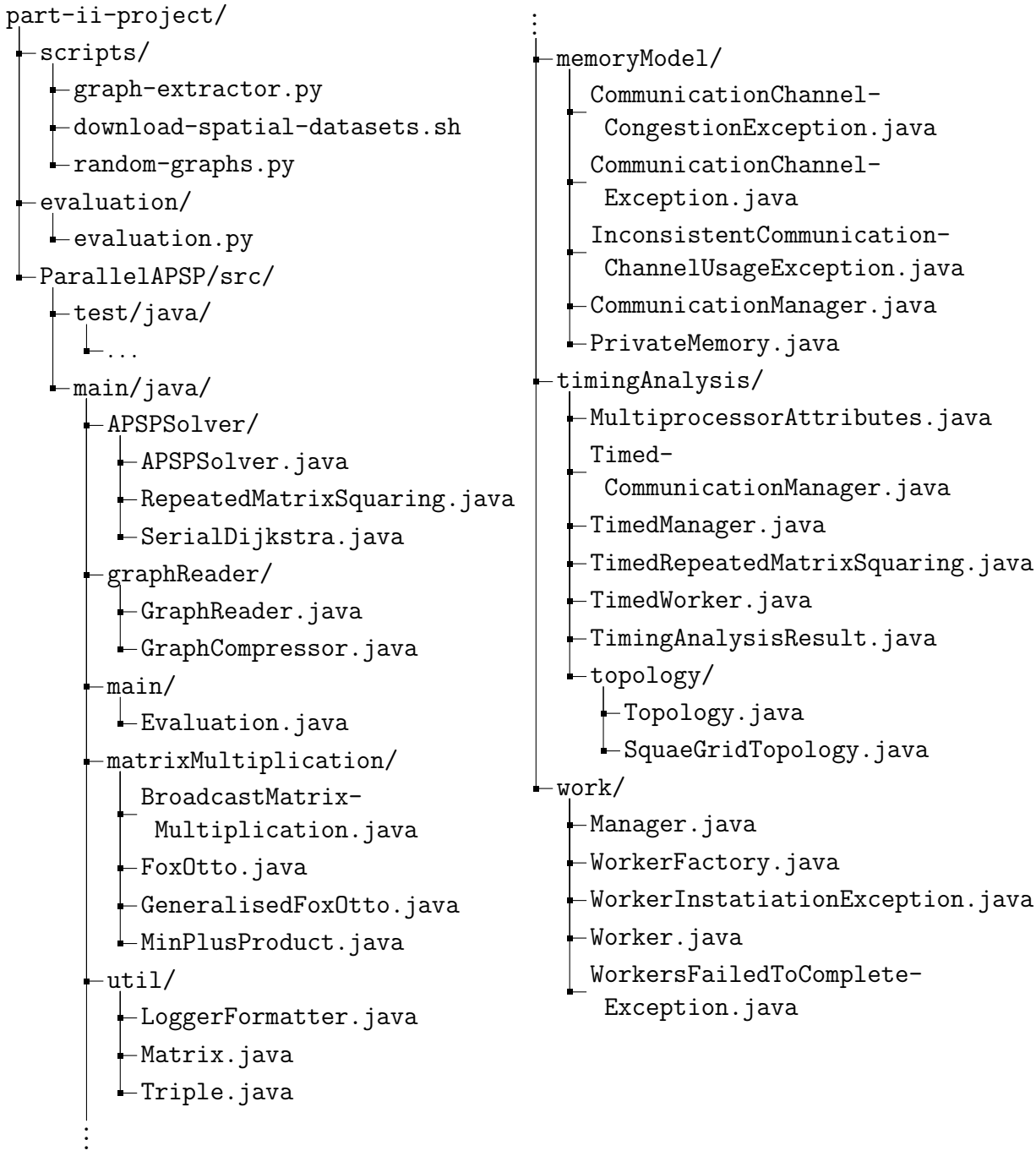


Figure 3.1: The directory structure of the code repository. Text in italics indicate Java interfaces, while regular text indicate classes. Subdirectories are Java packages. An overview of the unit tests in the `test/java/` directory is presented in figure 4.2 in the Evaluation chapter. The whole codebase, as presented here, contains 5877 lines of Java code, and 728 lines of Python code. The `work/`, `memoryModel/` and `timingAnalysis/` packages are elaborated in their respective subsections in section 3.2, and the `APSPSolver/` and `matrixMultiplication/` packages are described in section 3.3.

Choice of parameter p

To allow the evaluation to better analyse the practicality of using MatSquare for route-planning problems, we want the input graphs to have matching characteristics to real-world graphs. With a clever choice of p, we can achieve this.

Location	**Average vertex degree**
San-Francisco (SF)	2.549
North-America (NA)	2.038
City of Oldenburg (OL)	2.305
California (cal)	2.061
California (compressed)	2.945
City of San Joaquin (TG)	2.614

Table 3.1: Average vertex degree in various road-network datasets[3].

There are many characteristics of a graph, such as vertex- and edge-connectivity, betweenness centrality, and clustering coefficient. However, trying to generate new graphs with similar values for all of these metrics is another project itself. I have therefore only tried to replicate the metric, *average vertex degree*, which has some correlation to these metrics. I also want the random graphs to be connected, as most road-networks are. In the $G(n, p)$ random-graph model, the number of edges follows the binomial distribution:

$$\text{For vertex } v \in V, \;\; \deg(v) \sim \text{Binomial}(n-1, p) \tag{3.1}$$

with the average degree being $(n-1)p$. By modifying the graph generation algorithm to start with a *circular graph*[4], and including each remaining edge independently with probability

$$p = \frac{\text{desired average degree} - 2}{n-3}, \tag{3.2}$$

we get a connected graph where the average degree is as desired.

I chose the desired average degree based on that of the Californian road-network. However, since we will be using the compressed graph (see section 3.4), I used the average vertex degree of the compressed California graph instead, which was 2.945. In Figure 3.2, I have plotted an example graph generated through this method. I then generated 22 graphs of various sizes for use in evaluation[5].

3.2 Simulation of a distributed memory multiprocessor

The parallel system simulator, as shown in figure 3.3, is designed with a simple `Worker` interface in mind. To implement a parallel algorithm, the programmer describes how each processing ele-

[3]These values were computed using the `printSummary` method in the `GraphReader` class.

[4]Each vertex has exactly 2 neighbours.

[5]To get a sense of scaling for both small and large problem sizes, the graphs were of sizes: `10,20,30,40,50,60,70,80,90,100, 150,200,250,300,350,400,450,500,550,600,650,700`

[7]The graph was visualised using the `ForceAtlas2` library in python.

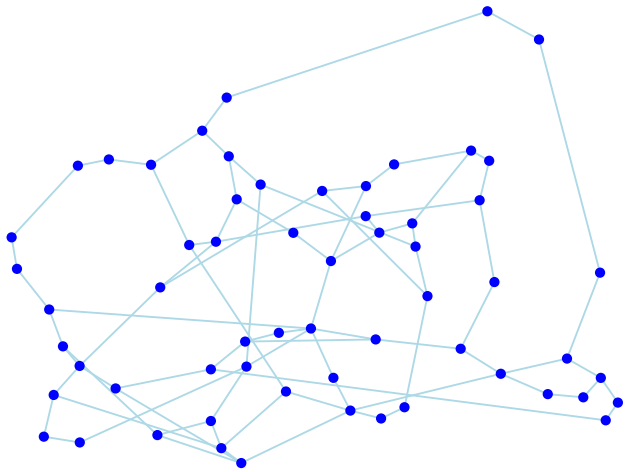


Figure 3.2: A visualisation of a graph generated with the modified Erdös-Renyi model. The graph has $n = 60$ vertices, and the length of the edges correspond to their weight[7].

ment (PE) should be initialised, and what computation and communication it should do in each of the phases—`communicationBefore(`ℓ`)`, `computation(`ℓ`)`, and `communicationAfter(`ℓ`)`—at iteration ℓ. This algorithm is then passed to the `Manager` which creates p^2 `Workers` using a factory, and the programmer just needs to call `Manager::doWork()` for all the workers to start executing. The execution of worker-phases is efficiently interleaves by the `Manager`, which uses the `CommunicationManager` to handle the messages passed between PEs. During execution, the computation time is directly measured, while the communication time is estimated using a mathematical model.

3.2.1 Work management

After describing an algorithm through the `Worker` interface, the programmer sets up simulated execution with the `Manager`. They specify the number of PEs, the problem size, the number of computation iterations and the input. When `Manager::doWork` is called, execution is simulated and the output can be retrieved with `getResult(label)`.

The work done by each simulated PE is specified in phases, as seen in the `Worker` interface in figure 3.3. If we consider a single PE in isolation, the order of these phases is described in algorithm 1. However, the work management is not so simple because the PEs interact during communication. As we saw in the previous subsection, the sent messages are put in a buffer and are not received until `CommunicationManager::flush` is called. Therefore, if worker i sends a message to worker j in their nth phase, the execution order dependency is

$$\text{phase } n \text{ of worker}_i \rightarrow \texttt{flush} \rightarrow \text{phase } n+1 \text{ of worker}_j. \tag{3.3}$$

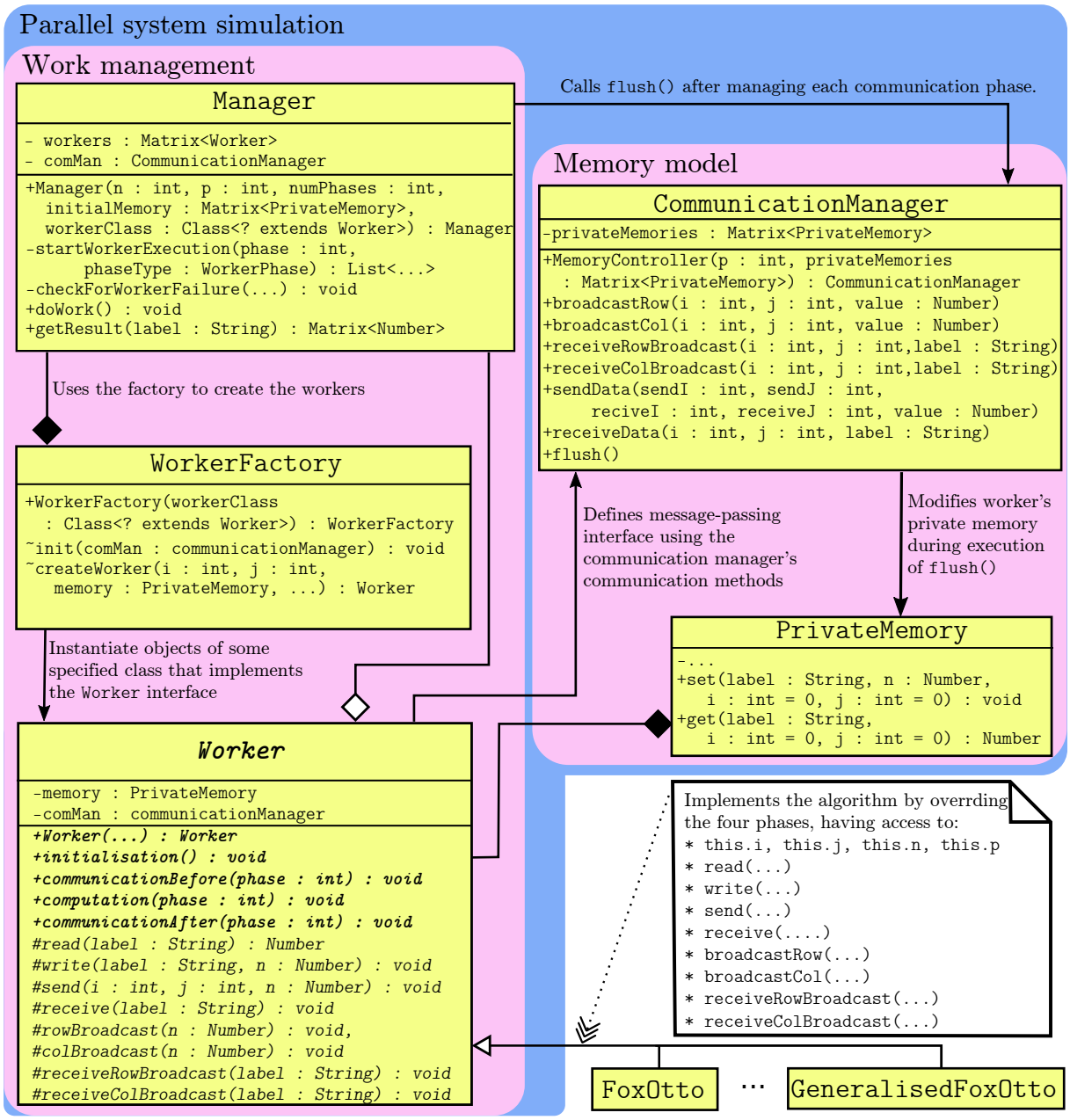


Figure 3.3: An overview of the main components of the simulator and how they interact. The `Manager` constructs the specified number of `Workers` based on some provided subtype, for example `FoxOtto`, using a `WorkerFactory`. When `doWork()` is called, it will then run the `Workers` through all of their communication- and computation-phases, and `flush()` the effects of their communication after each phase, which is done using the `CommunicationManager`. The `Worker` is an abstract class and provides a simple yet expressive interface for specifying parallel algorithms.

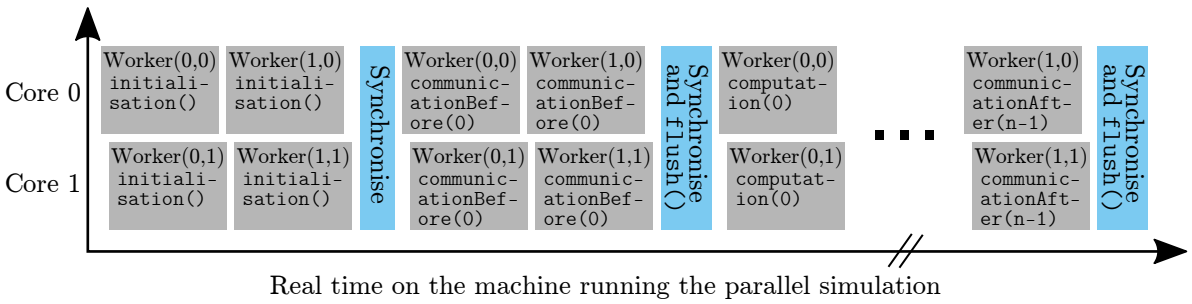


Figure 3.4: Example of multithreading when simulating a 4 processing elements on a host computer with 2 cores.

These constraints are satisfied by running all the workers through each phase before moving onto the next, as visualised in figure 3.4. This execution pattern also allows exploiting parallelism on the host machine for efficiency.

Algorithm 1: Execution of PE(i, j)

```
1 worker.initialisation();
2 for l = 0 ... number of iterations − 1 do
3 |   worker.communicationBefore(l)
4 |   worker.computation(l)
5 |   worker.communicationAfter(l)
6 end
```

Splitting the work into phases has many benefits. When investigating the different algorithms, as well as possible extension like parallel Floyd-Warshall, I noticed a clear distinction between computation and communication. Enforcing a separation between the phases would therefore not reduce the expressiveness. Instead, it allowed representing each phase as an independent `Callable` task, which has many benefits. Firstly, workers can be managed with higher-level constructs like a Java `ExecutorService` instead of having a Java `Thread` for each one. This prevents the JVM from creating more threads than what the host computer can efficiently manage[8] as the threads are reused for new computation, as shown in figure 3.4. As a result, we avoid a lot of the overhead associated with Java's `Thread` API. Another benefit is implicit synchronisation. Because of the data dependencies in equation (3.3), we do not need synchronisation primitives within the communication methods of each `Worker`. Instead, synchronisation and correct execution order is achieved through the scheduling of the work, as seen in figure 3.4. The third benefit of using work phases is allowing repeated execution of the same work, which gives a more accurate computation time estimates.

The `WorkerFactory` uses Java's reflection API to create instances of any implementation of the ***`Worker`*** interface. When constructing this factory, the programmer provides a `Class` object of the `FoxOtto` implementation, for example. The factory can then infer the appropriate constructor at runtime and create p^2 `FoxOtto`-workers, without needing a dedicated Fox-Otto worker factory. This simplifies the interface as implementing a ***`Worker`*** is everything needed to add a new parallel algorithm.

[8]This number is configurable, but since my Laptop can execute 8 threads simultaneously, I used this value.

3.2.2 Memory model

To make memory access simple, we use string labels. A PE can then for example store a value at location `"A"` and then retrieve it with the same label later. Additionally, for PEs handling a non-1 × 1-submatrix, the local memory is arranged in a `Matrix` where the PE can for example store values at location `(1,4,"A")` to associate it with an intermediate result $C_{1,4}$.

To send point-to-point messages, both the sender and recipient must call `sendData` and `receiveData`, respectively. The `receiveData(i, j, label)` method does not return a value, but instead modifies the recipient's memory at label `label`, but this change does not happen until `CommunicationManager::flush()` is called. When sending data with `sendData`, `broadcastCol` or `broadcastRow`, the data is buffered in the `CommunicationManager`. When `flush` is called, all the buffered messages are matched up with corresponding calls to `receiveData`, and the private memories of the recipients are modified by the `CommunicationManager`. To prevent race conditions when buffering data, fine-grained locking mechanisms are used within the `CommunicationManager`. An obvious alternative to this approach is making the `receiveData` method return the number received, which can be done by putting the recipient `Worker` to sleep until data has been received. Compared with this approach, my approach allows cleanly separating computation and communication. As we will discuss in the following subsection, this gives many benefits.

3.2.3 Estimating computation and communication times

Even if we are using a simulator, we want the execution time measurements to be as realistic as possible. This is achieved by measuring computation time directly and using a communication model proposed by Gergel for message-passing [7]. I first cover the theory and ideas behind how the measurements done, then go into how this is implemented in the `timingAnalysis` package.

Measuring computation time

We can measure the computation time directly. By taking the difference between the CPU time before and after the computation phase, we get a measure for how long the phase was. We also want to minimize the effects of running the code in a simulated environment. Compared to a real parallel system, where there is dedicated processor and private memory for each PE, the memory and processing power on the host machine is shared between all the simulated workers and various background processes. This can cause unwanted effects such as unexpected cache misses, due to other PEs' phases being simulated between the execution phases of a particular PE. Other effects include background processes temporarily using a high CPU load. To mitigate all this, we want to repeat each phase several times and average their computation time.

Mathematical model for communication

The communication time is estimated with a theoretical model. The time it takes to send a s-byte message can be modelled with $t = \alpha + \beta \cdot s$, which is a model that has been previously used to estimate communication time [7]. To simplify our model, I assume that the constants α and β are fixed for all point-to-point links. Incorporating our interconnect topology, if the shortest distance between two nodes i and j is $\delta(i, j)$, then the time it takes to send a message m from node i to j is

$$t(i, j, m) \triangleq \delta(i, j) \cdot (\alpha + \beta \cdot \text{size}(m)). \quad (3.4)$$

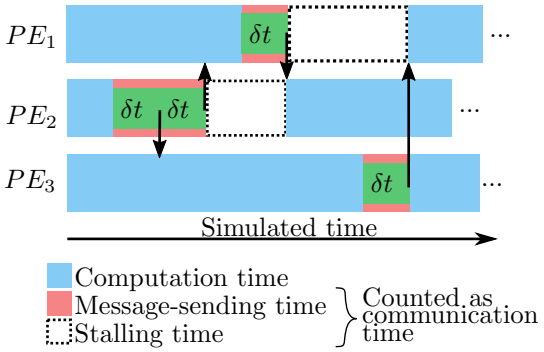


Figure 3.5: An example of the communication model based on equation (3.5). The "δt"s are the times it takes to send a message, as defined in equation (3.4).

Since each PE executes independently, if recipient j awaits data from sender i at some time, but the sender has not reached the appropriate communication phase by this time, the recipient must stall until i is ready and has sent the data. An example of this is shown when PE_1 sends a message to PE_2 in figure 3.5. To model this, let $T_i^{(n)}$ be the simulated time at which PE i starts executing phase n. I make the simplifying assumption that if the recipient has received all the data it expected by the time it reaches the corresponding communication phase, it can immediately start the next phase after sending off the data it needs to send. This happens for PE_3 in figure 3.5. With this in mind, if PE i is receiving data from a set of sender-PEs, S, and sending data to a set of receivers, R, at communication phase n, which starts at time $T_i^{(n)}$, then the next phase starts at

$$T_i^{(n+1)} = \max \left(\overbrace{T_i^{(n)} + \sum_{(j,m) \in R} t(i, j, m)}^{\text{All necessary data has already arrived}} \quad , \quad \underbrace{\max_{(j,m) \in S} \left(T_j^{(n)} + t(j, i, m) \right)}_{\text{PE } i \text{ must stall until data has arrived}} \right). \quad (3.5)$$

The `timingAnalysis` package

The functionality for measuring the computation time of each PE and estimating the communication time required for message-passing is implemented using the *decorator* pattern, where an overview of the components is shown in figure 3.6. This pattern made implementation more modular, allowing the core parts of the simulator to be developed before timing measurements were added.

The `MultiprocessorAttributes` class is used to store what constants we assume for the message-passing latency and bandwidth, α and β, both for point-to-point messages and for

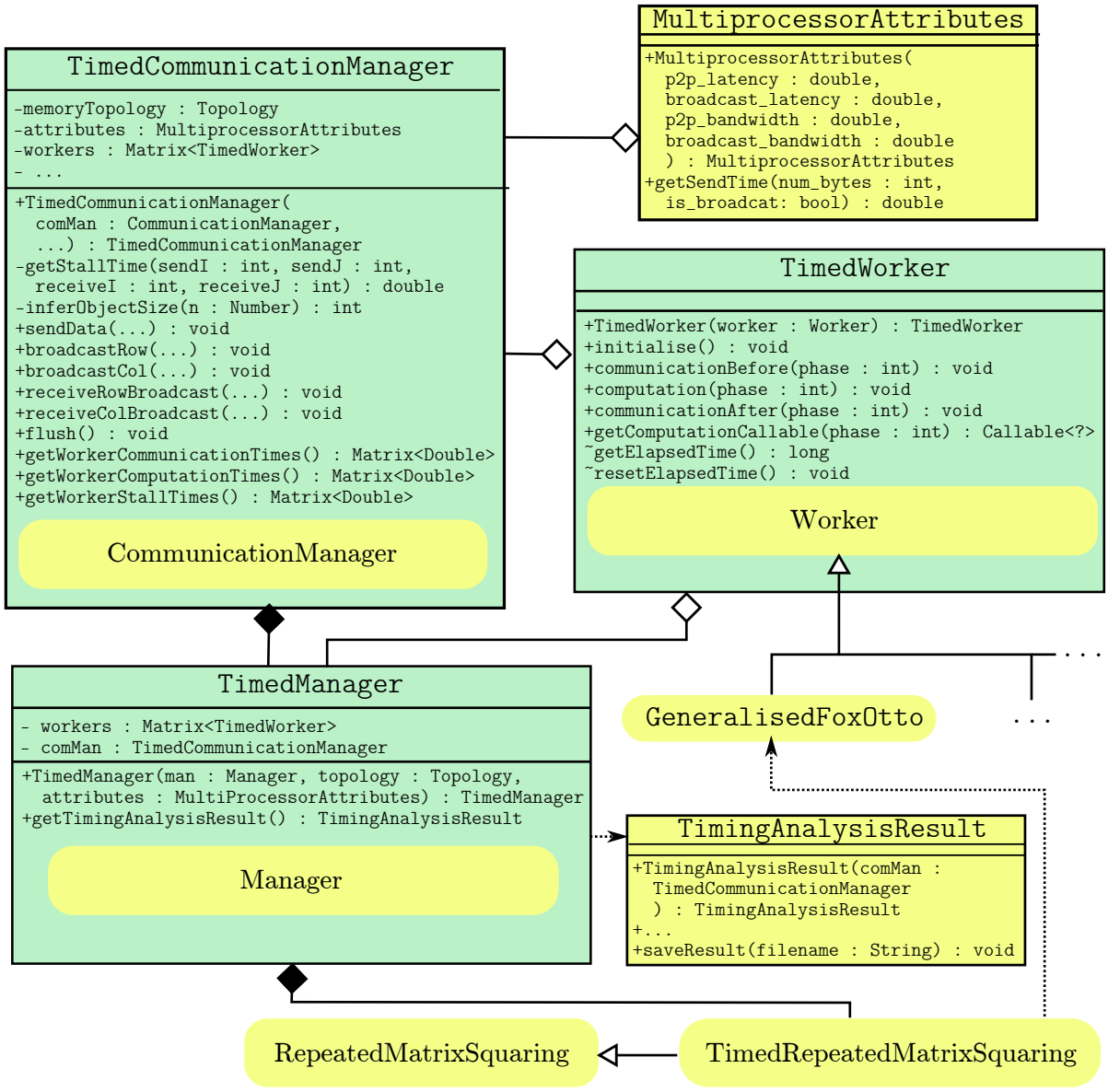


Figure 3.6: An overview of the components of the `timingAnalysis` package. Note that the `GeneralisedFoxOtto` and the `RepeatedMatrixSquaring` classes are not part of this package. Additionally, the green class diagrams with a yellow class inside it are short-hands for the decorator-pattern: The green classes extend the base class, shown in yellow, and contains a reference to it that is used to perform the original functionality in overridden methods in addition to the timing analysis behaviour.

broadcasting. We also implement the δ function using the `Topology` interface. Together, these two classes implement function t in equation (3.4).

Algorithm 2: Measuring computation time in `TimedWorker`

Data: Phase number l, *num. repeats*
Result: Computation time t

```
/* Enable read-only                    */
computation(l);
t_0 ← current time ;
for i = 0 ... num. repeats − 1 do
    computation(l);
end
t ← (current time − t_0) / num. repeats
/* Disable read-only                   */
computation(l);
```

Most of the functionality for simulating execution time is found in `TimedCommunicationManager`. Here, we maintain a matrix of the simulated times of each PE, with entries (i, j) initially corresponding to times $T^{(0)}_{(i,j)}$. These entries are then updates whenever we advance from phase (n) to phase $(n + 1)$, with different rules for the phase type. For computation phases, we simply increment the time with the measurement from the `TimedWorker`. For the communication phases, we use the formula in Equation 3.5, where the sets S and R are constructed by tracking calls to `sendData`, `broadcastRow`, `receiveData`, etc. We also only evaluate this formula when executing the decorated `flush()` method after each communication phase.

In `TimedWorker`, we use a clever trick to get more accurate measures for the computation time, as shown in algorithm 2. The read-only mode makes all side-effects of call to `Worker::store` be ignored. This is necessary to ensure the computation always follows the same control flow. By running the unit of computation once before starting the timing, the effects of cache misses are minimized as the used data will be in cache by the time we get to the first timed `computation(`l`)`. We also do several runs and average these to minimize the effect of background processes running on the host machine.

3.3 APSP via repeated matrix-squaring

We will now look at the *APSP via repeated matrix-squaring* algorithm, which I will abbreviate as *MatSquare*. Before describing its execution, I introduce some notation, and explain how the graph's adjacency matrix is used to find shortest paths by using an operation called the *distance product*. We then consider the predecessor matrix, and how it is used to reconstruct paths. When introducing the algorithm for finding the distance- and predecessor matrix, I will look aside from how the matrix-multiplication procedure is implemented, only focusing on its desired semantics. Afterwards, I describe a parallel implementation of the procedure, emphasising how it fulfils the semantics assumed.

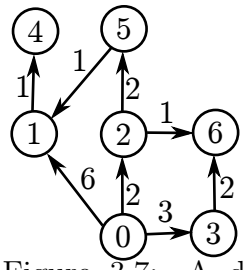


Figure 3.7: A directed graph with 7 vertices.

Notation

Before getting to the description of the algorithm, we must go through the notation I will be using. The parallel system has p^2 PEs, laid out in a square lattice of size $p \times p$. We are solving APSP for a graph with n vertices, so the input is a $n \times n$ adjacency matrix. Before executing the algorithm, the adjacency matrix has been split up into submatrices and been distributed evenly among the PEs. If p does not divide n, then we pad the adjacency matrix such that each PEs receives a submatrix of size $\lceil n/p \rceil \times \lceil n/p \rceil$. We refer to the height and width of this submatrix as the *submatrix size*, abbreviated n'. The padding happens by adding $n - \lceil n/p \rceil \cdot p$ vertices with no edges to the graph G. From here on, we will let n be the number of vertices in this padded graph, and the adjacency matrix A will also include inserted empty nodes. The goal of the algorithm is finding the distance- and predecessor matrix.

Definition 2. *A distance matrix M_{dist} is a $n \times n$ matrix, whose (i, j) entry is the length of a shortest path $i \rightsquigarrow j$ in graph $G = (V, E, w)$.*

Definition 3. *A predecessor matrix M_{pred} is a $n \times n$ matrix, whose (i, j) entry is the immediate prior node to j in a shortest path $i \rightsquigarrow j$ in graph $G = (V, E, w)$.*

We also refer to $M^{(x)}_{dist}$ and $M^{(x)}_{pred}$ as the corresponding matrices, but only considering paths of length *at most* x. This then gives that $M^{(n)}_{dist}$ and $M^{(n)}_{pred}$ are distance and predecessor matrices, respectively[9].

The MatSquare algorithm

This algorithm computes the distance- and predecessor matrix. Let A be the adjacency matrix for graph G. We want to add self-loops to each vertex, by setting $A_{i,i} \leftarrow 0$ for each $0 \leq i < n$, because it allows repeated distance products to capture shorter paths. For example, if A is the adjacency matrix for the graph in figure 3.7, $A^4_{0,1}$ would be the length of the path $1 \xrightarrow{2} 2 \xrightarrow{0} 2 \xrightarrow{2} 5 \xrightarrow{1} 1$, which in practice just contains 3 edges. We notice that this modified adjacency matrix fits the definition of $M^{(1)}_{dist}$. Additionally, computing the distance product between two matrices $M^{(x)}_{dist}$ and $M^{(y)}_{dist}$ gives a matrix $M^{(x+y)}_{dist}$ containing distances shortest paths of length at most $x + y$. As such,

Algorithm 3: MatSquare

Input : Adjacency matrix A
Result : Distance matrix M_{dist}, Predecessor matrix M_{pred}

```
1  M_dist ← W;
2  M_pred ← n × n matrix;
3  for (i, j) ∈ V² do
4  |   M_pred[i, j] ← i if W_{i,j} < ∞ else j;
5  end
   // Repeated squaring
6  for x = 1 ... ⌈log₂ n⌉ do
7  |   M_dist ← M_dist ⊗ M_dist;
8  |   W ← witness matrix for above ⊗;
9  |   for (i, j) ∈ V² do
10 |   |   if W_{i,j} ≠ j then
11 |   |   |   M_pred[i, j] ← M_pred[W_{i,j}, j];
12 |   |   end
13 |   end
14 end
```

[9] As long as the graph G does not have negative cycles

one way to compute the distance matrix is

$$M_{dist} = M_{dist}^{(n)} = \underbrace{M_{dist}^{(1)} \otimes \cdots \otimes M_{dist}^{(1)}}_{n-1 \text{ times}} = \underbrace{A \otimes \cdots \otimes A}_{n-1 \text{ times}} \tag{3.6}$$

However, we can reduce the number of distance product matrix multiplications from $\Theta(n)$ to $\Theta(\log n)$ by squaring A $\lceil \log n \rceil$ times, which is what happens in the `for`-loop in algorithm 3. With this transformation, we may overshoot by finding $M_{dist}^{(n')}$ for $n' > n$, but this will be the same matrix because all the shortest paths cannot be of length longer than n.

Reconstructing paths We have now described how to compute the length of the shortest paths, but we would also like to reconstruct the list of nodes that make up these paths. When computing $M_{i,j} \leftarrow \min_{0 \le k < n}(M_{i,k} + M_{k,j})$ as part of the distance product $\otimes$, the operation corresponds to finding the optimal intermediate node k such that $i \rightsquigarrow k \rightsquigarrow j$ is the shortest path from i to j. If we already know the predecessor of j in the path $k \rightsquigarrow j$, then the predecessor of j in path $i \rightsquigarrow k \rightsquigarrow j$ must be same. This allows us to correctly update the predecessor matrix after each distance product. Formally, we call these intermediate nodes for *witnesses*, where

Definition 4. *A witness matrix W for a distance product $M \otimes M'$ is a $n \times n$ matrix, whose entries (i, j) are witnesses*

$$w_{i,j} = \operatorname*{argmin}_{0 \le k < n}(M_{i,k} + M'_{k,j}). \tag{3.7}$$

These predecessor updates happen on lines 8–13 in algorithm 3. We also note that the witness k may be equal to j, in which case the path $k \rightsquigarrow j$ is a self-loop. This will obstruct the path reconstruction algorithm, so we want to keep the predecessor from path $i \rightsquigarrow k = j$ in those cases, hence the `if`-statement on line 10.

The algorithm to reconstruct the paths is simple once we have computed the predecessor matrix M_{pred}. Given a query $i \rightsquigarrow j$, we start with a list of vertices $[j]$, and repeatedly prepend the predecessor of the first vertex. We stop when we arrive at i or find a self-loop. In the latter case, the vertices are not connected. This is also why we initialise the predecessor entries $M_{pred}^{(x)}[i, j]$ to j if there are no paths $i \rightsquigarrow j$ of length x. An example of this algorithm is shown in appendix B.3.

Parallelising MatSquare with FoxOtto

We will now look at how to parallelise the matrix multiplication step and associated assignments in lines 6–14 of algorithm 3. I describe a parallel algorithm to perform the operations of a single iteration of the `for`-loop on lines 7–13 in algorithm 3, and this procedure will be executed $\lceil \log_2 n \rceil$ times. In each iteration, p^2 PEs execute algorithm 4 in parallel. To simplify the explanation, we consider what happens in the `for`-loop of algorithm 3 using slightly different notation: Let A be the left matrix of the distance product and B be the right matrix, such that both matrices are equal to M_{dist} in line 7. We also let P be the predecessor matrix M_{pred}. We

think of these three matrices as the input. After the parallel procedure, the distance product and the predecessor matrix should be stored in M_{dist} and M_{pred}, respectively, just like in algorithm 3.

To parallelise the procedure, we distribute the input A, B, and P evenly among the p^2 PEs by dividing them up into submatrices of size $\lceil n/p \rceil \times \lceil n/p \rceil$. For example, if we use a 2×2 lattice arrangement and distribute the adjacency matrix of graph in figure 3.7, the PE with id $(0, 1)$ would receive entries $[0 \ldots 3, 4 \ldots 7]$ of matrices A, B, P. It would also be responsible for computing the submatrices $M_{dist}[0 \ldots 3, 4 \ldots 7]$ and $M_{pred}[0 \ldots 3, 4 \ldots 7]$. As submatrices of A, B and P are shifted around during communication, I use the notation A', B' and P' to refer to submatrices the PE currently holds, but that might be shifted around. Also, M'_{dist} and M'_{pred} refer to the submatrices a PE is responsible for computing.

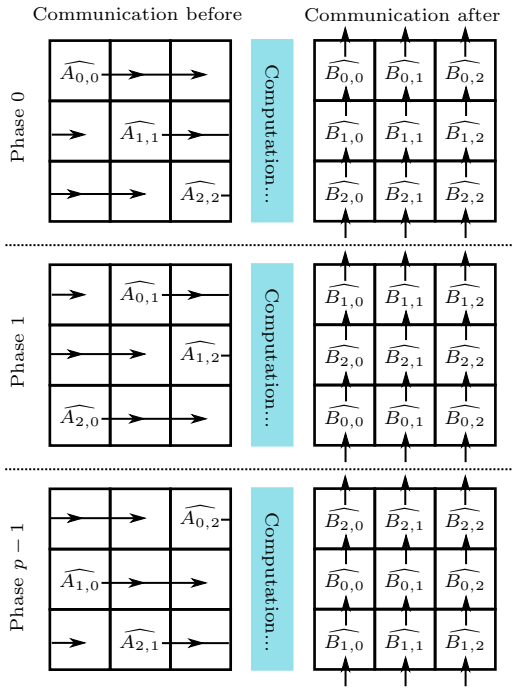


Figure 3.8: Message-passing pattern used in the Fox-Otto technique.

If we consider the computation of an entry (i', j') in the distance product in equation (2.4), the order of the products $A_{i',j} \cdot B_{k,j'}$ does not matter because the min-operator is commutative. By first considering the data that is local to each PE first, and then considering other ks as data is shifted around, the PEs can simultaneously perform minimisations without requiring more than $O(\lceil n^2/p^2 \rceil)$ memory. Fox et al. describe a data movement technique for doing this [5]. As seen in figure 3.8, this technique consists of broadcasting the submatrices of A along the rows before partially computing the min. After computation, the submatrices of B ared shifted upwards to the next PE. This pattern ensures that in the computation phases, all the PEs have access to matching submatrices of A and B. These communication phases are implemented in lines 9–11 and 26–27 in algorithm 4.

Since each PE $p_{i,j}$ is responsible for computing a $n' \times n'$ submatrix of M_{dist} and M_{pred}, we have two `for`-loops on line 12 that iterate all the entries (i_2, j_2) in these submatrices. In each of these iterations, we loop through the n' witnesses on line 13 and compare the new distance term with the current minimum distance on lines 17–18 in accordance with the definition of the distance product. We can ascertain ourselves that $A'[i_2, m]$ and $B'[m, j_2]$ contain matching elements of the matrices A and B by working through the data movement pattern in figure 3.8. In fact, it can be shown that on line 17, we are computing $A[i', k] + B[k, j']$ with i', j' and k as defined on lines 14–16. Therefore, the witness will be k, which makes the

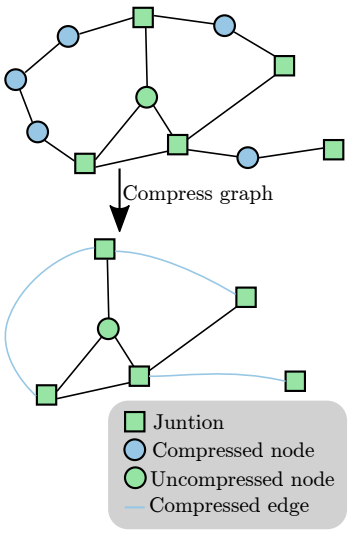


Figure 3.9: Graph before and after compression.

Algorithm 4: Generalised Fox-Otto execution at processing element $p_{i,j}$

Data : A', B', P'
Parameter : problem size n, PE grid size p, subMatrixSize $n' = \lceil n/p \rceil$,
Result : M'_{dist}, M'_{pred}

// ** Initialisation phase **
$A'_{const} \leftarrow A'$;
$M'_{dist} \leftarrow n' \times n'$ matrix;
$M'_{pred} \leftarrow n' \times n'$ matrix;
for $0 \leq i_2, j_2 < n'$ **do**
  $M'_{pred}[i_2, j_2] \leftarrow j$;
  $M'_{dist}[i_2, j_2] \leftarrow \infty$;
end
for *phase number* $l = 0$ *to* $p - 1$ **do**
  // ** CommunicationBefore phase l **
  if $j = (i + l) \mod p$ **then**
    // Received as a submatrix A'
    broadcastRow(A'_{const});
  end
  // ** Computation phase l **
  for $0 \leq i_2, j_2 < n'$ **do**
    // We start with m s.t. $k = i'$ at $l = 0$, which causes shorter
    // paths from the previous squaring to be considered first.
    // This is necessary to get the predecessor right.
    for $m = i_2, i_2 + 1, \ldots n' - 1, 0, 1, \ldots, i_2 - 1$ **do**
      // In this iteration, we compute A[i', k] + B[k, j'], where
      $k \leftarrow (n' \cdot (i + l) + m) \mod n$;
      $i' \leftarrow i \cdot n' + i_2$;
      $j' \leftarrow j \cdot n' + j_2$;
      // Distance product
      $d_{new} \leftarrow A'[i_2, m] + B'[m, j_2]$;
      if $d_{new} < M'_{dist}[i_2, j_2]$ **then**
        $M'_{dist}[i_2, j_2] \leftarrow d_{new}$;
        // If $k = j'$, the path $k \to j'$ will be a self-loop, so
        // we should not update the predecessor
        if $k \neq j'$ **then**
          // The cell $P'[m, j_2]$ holds $P_{k,j'}$
          $M'_{pred}[i_2, j_2] \leftarrow P'[m, j_2]$;
        end
      end
    end
  end
  // ** ComunicationAfter phase l **
  send($p + i - 1 \mod p$, j, B');
  send($p + i - 1 \mod p$, j, P');
end

Road network	Nodes	2-degree nodes	Size reduction
California	21048	19683	×15.4
San Francisco	174956	29627	×1.2
North America	175813	166687	×19.3
City of San Joaquin	18263	3760	×1.26
City of Oldenburg	6105	3232	×2.12

Table 3.2: Possible size reduction by removing 2-degree nodes from different road networks

predecessor update logic on lines 20–22 equivalent to what is done in lines 11–13 in algorithm 3.

3.4 Graph compression

Most road networks are sparse, so many of the vertices will just have two neighbours. This can be used to reduce the amount of computation as there is only one path across a segment of two-degree nodes. In this section, I present an algorithm for compressing segments of two-degree nodes into a single edge. As we see in table 3.2, this can significantly reduce the problem size of real-world road networks, especially the Californian road network.

We compress the graph by removing all nodes with exactly two neighbours and their edges. We refer to a contiguous sequence of such compressed nodes as a *(compressed) chain*, and compensate for its removal by adding a new edge between the *junctions* of the compressed chain, as shown in Figure 3.9. The weight of the new edge is the sum of the weight of the edges removed. We do this for all chains containing at least one node of degree two. There are two edge-cases that may occur during the compression. Firstly, if the two junctions are the same node, I let a 2-degree vertex adjacent to the junction be the second junction. This avoids introducing more edge-cases in the path-reconstruction step later. Secondly, if we have a two-degree-node path $p \to n_1 \to \cdots n_l \to q$ between two junctions p and q, and there is an edge $(p, q) \in E$, then the compressed graph will be a have two different edges between p and q. In this case, we only keep the edge with the lowest weight, discarding the other.

To get any benefit from running some APSP algorithm, like the MatSquare algorithm, on the compressed graph, we must be able to map path queries for the original graph onto the compressed graph, and then map back the result. To do this, extra bookkeeping during graph compression is needed. For each compressed node, we store a map to its two junctions, the list of nodes in the paths to the junctions, and the length of these paths. Additionally, with each compression edge, we store the list of compressed nodes.

To answer an APSP query $p \rightsquigarrow q$, and reconstruct its path, consider the original graph in Figure 3.10. We have 5 cases, each of which gives different queries to the original graph:

1. Nodes p and q are on the same compressed chain, in which case no queries are made. Instead, we compute the path using the list of vertices associated with the compressed chain. This case corresponds to $A = C$ and $B = D$ in figure 3.10.

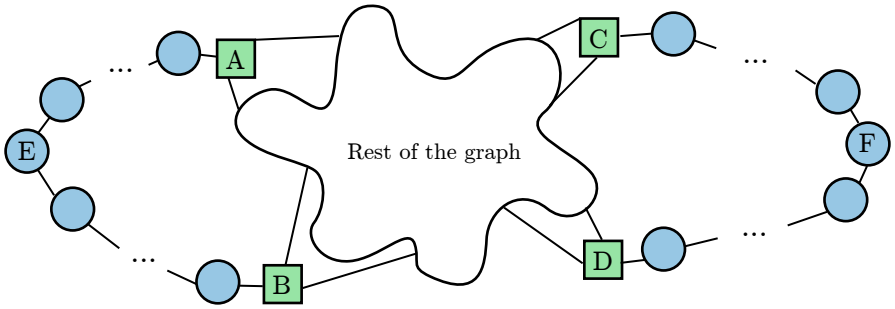


Figure 3.10: Without loss of generality, if the original graph has at least one node with degree 2, we can visualise it like this.

2. We have $p = E, q = F$. We compute the four paths, $p \rightsquigarrow \{A, B\} \rightsquigarrow \{C, D\} \rightsquigarrow q$, and use the shortest one. This gives four queries on the compressed graph.
3. We have $p \in \{A, B\}, q = F$. We compute the two paths, one going through junction C and the other through D. The queries on the compressed graph are $p \rightsquigarrow C$ and $p \rightsquigarrow D$.
4. We have $p = E, q \in \{C, D\}$. This is symmetric to case (3).
5. We have $p \in \{A, B\}, q \in \{C, D\}$. Both nodes are on the compressed graph, so we query $p \rightsquigarrow q$.

For all of the above cases, we get back a path on the form:

$$p \overset{\text{nodes just in original graph}}{\rightsquigarrow} J_1 \overset{\text{nodes just in compressed graph}}{\rightsquigarrow} J_2 \overset{\text{nodes just in original graph}}{\rightsquigarrow} q, \tag{3.8}$$

where any of the three subpaths may be empty. To map this back to a path in the original graph, we must expand all the edges in the path $J_1 \rightsquigarrow J_2$ to a path in the original graph. This is done by going through each edge, and swapping the compressed chain out for the list of edges that were compressed.

3.5 Summary

In this chapter, I explained the components implemented throughout the project. I discussed the datasets used as input. Then I went into depth on how I implemented a simulation of a parallel system, which met the requirements outlined in section 2.4. The timing-estimation-functionality exceeded what was initially proposed, and the interface for writing parallel programs was very successful. I then explained the MatSquare algorithm, by first considering a high-level overview of it, followed by an in-depth description of how the computation was parallelised. Lastly, I looked at an optimisation that could significantly reduce the computation time required for solving APSP on sparse graphs.

Chapter 4

Evaluation

In this chapter, I review the software produced in the implementation phase. I relate systems and algorithms built to the success criteria in the project proposal, and describe how the work done exceeds what I initially set out to do. This is done by looking at the MatSquare algorithm, the parallel system built, and the optimisations applied. I review the tests I conducted to assert correctness. Then I proceed to discuss the advantage of parallel computation for solving APSP, where I demonstrate that the benefit is large, even for different types of parallel computers.

4.1 Parallel MatSquare

I successfully implemented the MatSquare algorithm. I initially set out to "*[implement] an algorithm based on matrix multiplication that can find the length of the shortest path between all pairs of nodes in a graph, and it is able to give the list of nodes that make up such paths.*" This resulted in the MatSquare algorithm, which is indeed based on matrix multiplication, as shown in lines 7–8 in algorithm 3. To assert the correctness of the path lengths and the path reconstruction, I manually computed the results for two small graphs, then compared it with the algorithm's output. Additionally, I took a subsection of one of the real-world graph datasets[1], and tested the path lengths and reconstructions on all

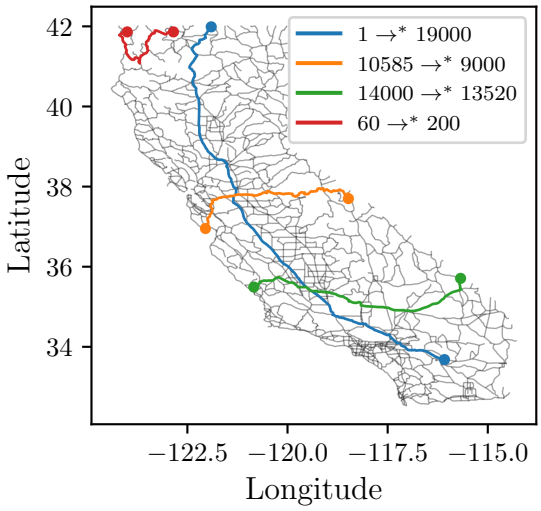


Figure 4.1: Four example shortest-paths in the Californian road network ($|V|$= 21048, $|E|$= 21693). These paths consist of 575, 328, 272, and 207 nodes, respectively.

[1]I used the `graph-extractor.py` script to extract all the nodes and vertices within a fixed distance away from a point in the centre of a city in the Oldenburg city road network dataset. This gave a graph with 430 nodes and 476 edges.

$|V|^2$ pairs. To do this, I implemented Dijkstra's algorithm with path reconstruction, which is a easier to prove the correctness of compared to the MatSquare algorithm. I then ran this and my algorithm and asserted the equality of the results for all 430^2 vertex-pairs. We can see the results of these tests in `RepeatedMatrixSquaringTest`s in figure 4.2. They were done for both the regular and generalised version of the algorithm.

I also successfully extended the project with the graph compression optimisation. Arbitrary undirected graphs can be compressed through removal of two-degree nodes, which I have tested by comparing the output to manually-compressed graphs for some example graphs. The algorithm for mapping APSP queries to the compressed graphs and mapping the results back is also correct. I verified this by comparing the paths found when compressing the graph to the output from MatSquare running on the same graph, but uncompressed. I did this in a unit test in figure 4.2, and found that it could correctly reconstruct the list of vertices in all 430^2 paths in a large graph. To further demonstrate that paths are reconstructed properly when mapping results from the compressed graph to the uncompressed one, I ran four example queries on the Californian road network and plotted the list of vertices returned in Figure 4.1, using additional data about each vertex's geographical position. Solving APSP on such a large graph would be infeasible to do quickly without the graph compression optimisation.

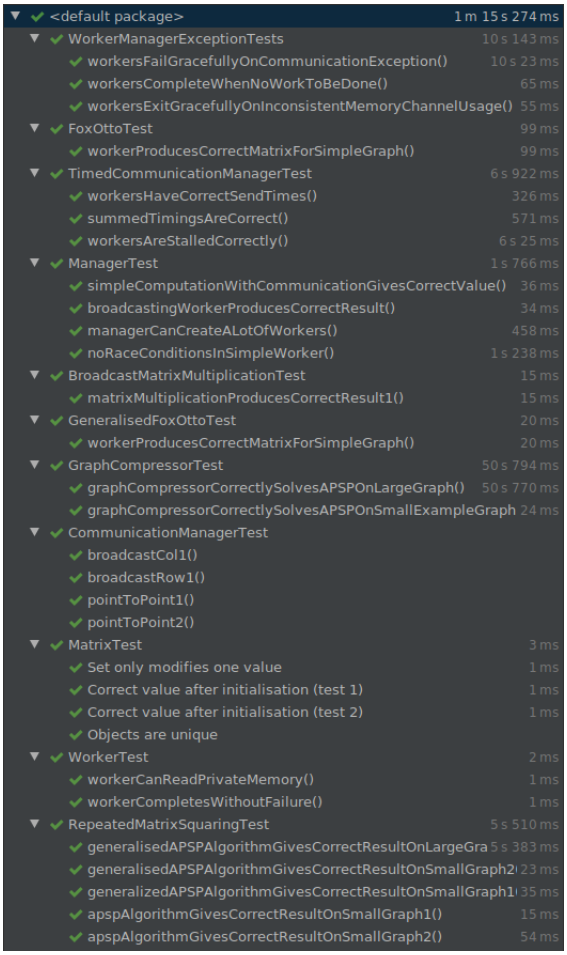


Figure 4.2: A screenshot showing all the unit tests written and that they passed.

I set out to "*... minimise the amount of data movement between processing elements, which is done by using techniques such as Fox-Otto's algorithm*" in the matrix-multiplication routine of the MatSquare algorithm. The data movement pattern used is based on the technique Fox et al. used [5]. As we see in algorithm 4, this makes all the PEs have exactly the data that they need at each computation phase, after sending a packet across a single interconnect channel, and possibly an uncontested row-broadcast channel as well. This is clearly minimal data movement if we are not allowed to utilize shared memory and each PE do not have sufficient private memory to store the whole input.

4.2 Parallel simulation

The parallel simulation met all the requirements laid out in section 2.4, which results in "... *a simulated massively parallel processor, where each processing element can send data to each other through simulated interconnects.*" When creating the `Manager` and `TimedManager`, I can configure the number of PEs and the interconnect topology, respectively. The `Worker` interface for writing parallel programs is simple to use, but has expressive power beyond what I required. It also cleanly threw useful exceptions such as `InconsistentCommunicationChannelUsageException` if the programmer did not correctly match up `send`s and `receive`s. The interface and its capability has been thoroughly tested through unit tests that cover usage much more complex than the what was needed by algorithm 4. The computation time measures reduce the amount of noise to give better measurements. Additionally, a MIMD model is used for the communication time, which is more complicated to implement compared to a naïve SIMD model, but with the benefit of giving a more realistic simulation. Lastly, the simulator runs in parallel by simultaneously executing worker-phases. This sped up simulation a lot, as computing the results for the compressed Californian road network for instance took about 25 minutes and the CPU usage was close to 800% while doing so[2], showing the parallelisation was successful.

I also successfully "*[p]arallelised the matrix multiplication routine of the [MatSquare] algorithm to run on [the parallel system simulator]*". This was achieved with the `FoxOtto` and `GeneralisedFoxOtto` classes, which use the `Worker` interface to create a parallel algorithm where coordination happens through message passing. The `Manager` allows intermediate results from each PE to be fetched after each phase, which I used in my unit tests to verify that the distance product was computed correctly on test matrix inputs. Another justification for the correctness of the parallel matrix multiplication step is the APSP unit tests. For these, I used the parallel matrix multiplication step instead of a serial one, and it is unlikely for the shortest paths to be correct if there are any faults in the parallel distance product subroutine.

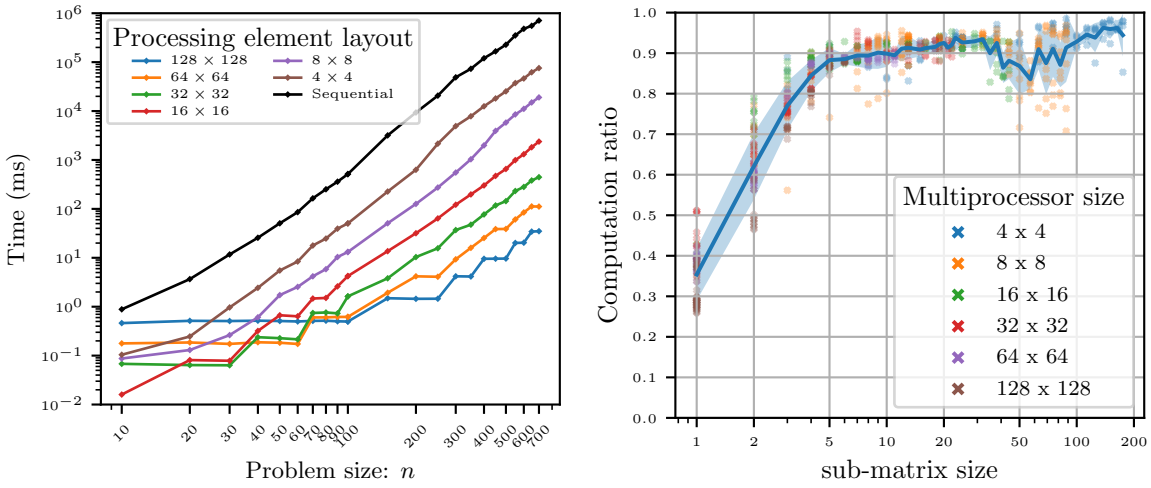
4.3 MatSquare timing measurements

Recalling the aim laid out in section 1.1.3 we want to evaluate the benefit of parallel computation on a wide range of parallel systems. With this purpose, I justify my choice of communication parameters. Then I plot and discuss their effect on both the total execution time and the parallel efficiency achieved on a wide range of input graphs with real-world characteristics.

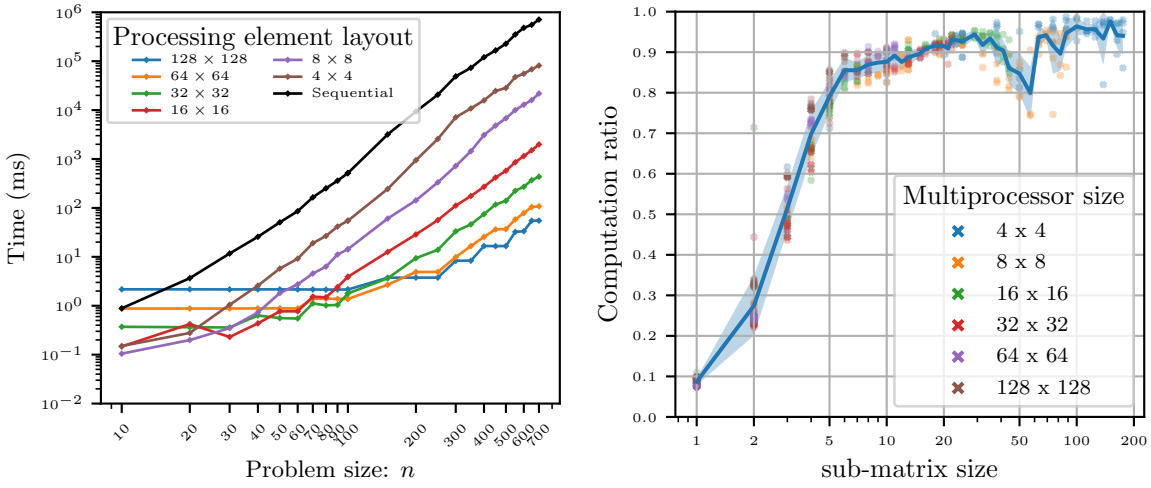
4.3.1 Choice of communication parameters

By basing the communication constants on real hardware, we can draw more applicable conclusions. For the parallel system simulator to estimate the message-passing delay, we must

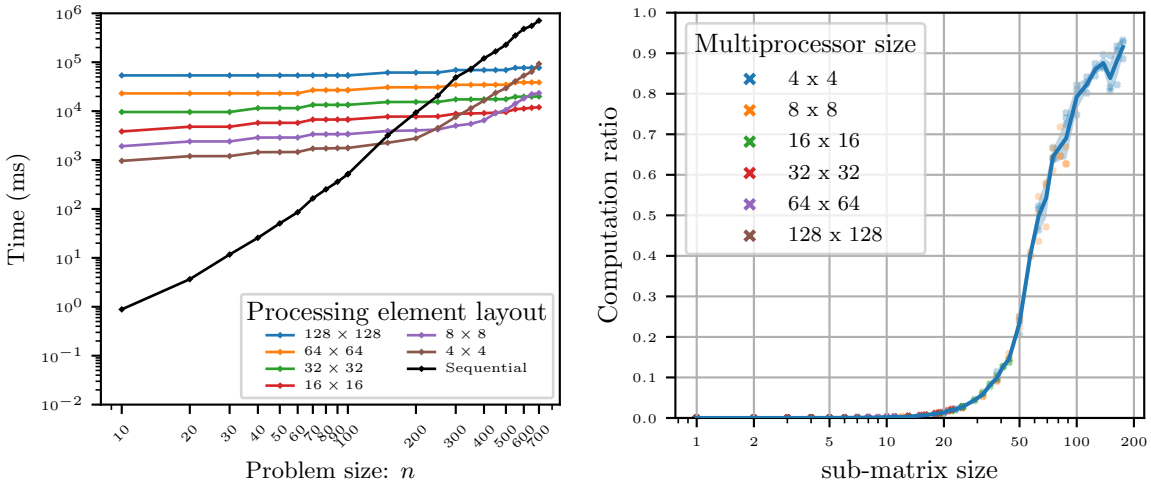
[2] I inspected this using `htop`



(a) Multi-core processor (Sandy Bridge)



(b) Supercomputer (Sunway TaihuLight)



(c) Distributed computer (Internet)

Figure 4.3: To the left, the total execution time of MatSquare is plotted for different graphs with $n = |V|$ vertices. To the right, I group computation ratios by the size of the sub-matrix each PE is responsible for. Confidence intervals are omitted from the left plots because they are mostly negligible, but can be found in section B.2 in the appendix.

specify the latency and bandwidth. As we reviewed in section 2.1, there are many classes of parallel computers, ranging from multi-core processors to different computers communicating over the Internet. To get a sense of what the benefit of parallelism is across this whole range, I consider the two extremes and a supercomputer that lies in-between. For each class, I base the communication constants of a real-life system within the class. The values chosen are summarised in Table 4.1.

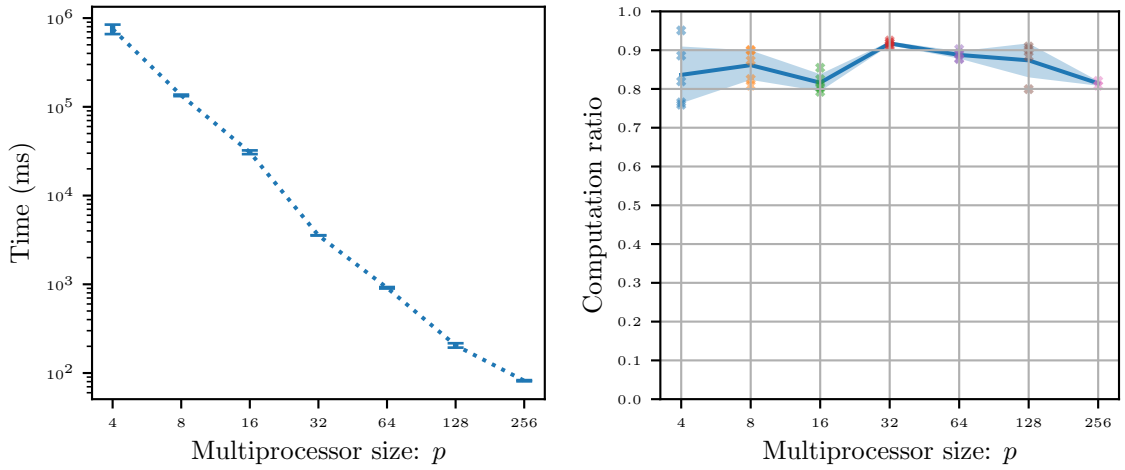
Multi-core processor The Sandy Bridge microarchitecture supports multiple cores which execute in MIMD fashion, and the L3 cache is shared across all cores while the lower-level caches are local to each core [13]. For one core to send data from its private L1 cache to another core's L1 cache, we must look at the round-trip time of the cache coherence protocol because the receiving core must message the sending core and wait for the cache line to come back. Campanoni et al. have measured the latency of this, and found that for the Sandy Bridge microarchitecture, the round-trip latency is about 107 clock cyles 80% of the time [2]. I make the simplifying assumption that the latency is always 107 cycles, and the latency is independent of the processor's clock frequency. In the ring interconnect used to implement cache coherency in the Sandy Bridge microarchitecture, has a bandwidth of 32 bytes per cycle. I will assume this is is the available bandwidth for message-passing.

Supercomputer For the supercomputer parallel system, I have based by constants on the Sunway TaihuLight system, which consists of several SW26010 processors connected together through a system interface with a bidirectional bandwidth of 16 GB/s and a latency around 1 µs [4]. Each SW26010 processor consists of 256 computer processing elements which can themselves communicate with lower latency and higher bandwidth, and they also run at a lower clock frequency than that of my laptop. I do not simulate this memory hierarchy, but instead assume that each individual PE is connected through the system interface used on the Sunway TaihuLight system, and executes instructions at the same clock frequency as my laptop. I also half the bandwidth because messages-passing is one-directional in my architecture assumptions in section 2.3.

Distributed computer For this parallel system, I am assuming that we have some set of PEs that are distributed geographically across Europe, and they communicate by sending packets over the Internet. This kind of system is very scalable as it is easy to add new PEs, but we expect the communication constants to be higher. Within Europe, ICMP packets have been measured to have an average round-trip time latency of under 15 ms, but I use a generous upper-bound of 30 ms because other continents might have higher latencies [15]. I also use the round-trip time to allow communication to happen with protocols where acknowledgement messages are sent as well, such as TCP. For the bandwidth, I used the regional average broadband speed across Western Europe, which is 90.56 Mbps [8].

Parallel system	Latency	Bandwidth
Multi-core processor	107 clock cycles[3]	32 Bytes per cycle[4]
Supercomputer	1 µs	8 GB/s
Distributed computer	30 ms	11.25 MB/s

Table 4.1: The communication constants used in my timing analysis of the MatSquare algorithm. Note that since each message consists of two or fewer `Doubles`, the latency will be the dominating contributor to the communication cost, not the bandwidth.



(a) Total time

(b) Computation ratio

Figure 4.4: Execution time measurements and computation ratio achieved when running MatSquare on the compressed Californian road network ($|V|= 1408$). I used communication constants based on a multi-core processor (Sandy Bridge).

4.3.2 Advantage of parallel computation for solving APSP

In figure 4.3, we see the scaling of the total execution time and the computation ratio when running MatSquare on different problem sizes. I start this subsection by discussing the general trends in the total time scaling in figures 4.3(a) and 4.3(b). Then I relate the plots to the parallel efficiency, and demonstrate the final success criteria while discussing the benefit of parallel computation on different machines and problem sizes.

For the multi-core and supercomputer configuration, the total execution time does not increase, or increases step-wise, for small problem sizes, especially for large PE layouts. In the flat portions, we have enough PEs for each cell in the input matrix, so increasing the problem size does not cause more computation per PE. The step-wise increase is caused by the input padding: To map for instance a problem of size $n = 200$ onto 64×64 PEs, we must pad the input until the size is a multiple of 64, which can cause two different problem sizes to be mapped to the same size, causing a flat segment.

As we get to larger problem sizes, the total times in figures 4.3(a) and 4.3(b) start to even out

[2]With the clock frequency of my system, this corresponds to 46.52 ns
[4]With the clock frequency of my system, this corresponds to 68.55 GB/s

and approach linear in the log-log-scale. Comparing the trends for the different values of p, we decrease the required computation time by a constant factor whenever we multiply the number of PEs by 4. We also see this for the compressed Californian road network in figure 4.4(a). This tells us that for large problem sizes, there's always a benefit to adding more computation power as this reduces execution time, even with a communication overhead.

We demonstrate the final success criteria by considering the parallel efficiency. As we saw in equation (2.3), the parallel efficiency is equal to the computation ratio. We can therefore think of the y-axis in three right-column plots in Figure 4.3 as being the parallel efficiency. With the multi-core processor and supercomputer configurations, we quickly achieve $\varepsilon > 0.9$ and it even reaches above 95% eventually. This means the MatSquare algorithm is able to efficiently utilize the available processing power on a parallel system without communication being a bottleneck. The last success criteria is therefore achieved: "*The evaluation of the algorithm demonstrates that parallel computation gives a high parallel efficiency for solving APSP*". We also see a parallel efficiency $\varepsilon > 0.8$ in figure 4.4(b).

In figure 4.3(c), we see that for a distributed computer, there is little benefit to parallel computation compared with serial for small problem sizes. When there are just a few µs of computation before we need to send a packet across Europe, there is a very large communication overhead, which we see from the near-zero computation ratio for sub-matrices of size less than 20. This is also why most of the execution time plots are flat, since the communication is dominating. However, even with such a large message-passing overhead, we eventually achieve a high parallel efficiency, but this only happens for the 4×4 and 8×8 arrangements for problem sizes where $n \leq 700$, but we can expect other arrangements to also get to this point with larger problems. Also, as we reach the point where computation start to dominate, we total execution time starts following a trend similar to that in figures 4.3(a) and 4.3(b). Therefore, we would expect the 128×128 configuration to overtake all the smaller arrangements for very large problem sizes.

4.4 Summary

In this chapter, I showed how the project met and exceeded the success criteria. The parallel system simulation was showed to work well through unit tests, and the interface it provides was very easy to work with. I demonstrated the correctness of the main algorithm of the project, and used timing measurements to prove the benefits of parallel computation for solving APSP on real-world-like graphs on a wide range of different parallel systems.

Chapter 5

Conclusions

In this dissertation, I have described the design, implementation and evaluation of a parallel system simulator and an algorithm for solving APSP. The project was a great success: The simulator was simple, expressive and efficient and the MatSquare algorithm and its extensions gave correct outputs for large graphs. Furthermore, the evaluation covered an elaborate qualitative analysis on the benefit of parallel computation, which contributes to the original aims of the project—investigating the benefit of parallel computation and its extent on different parallel system and inputs.

5.1 Results

As shown in section 4.2, the project has met all the success criteria related to developing an efficient simulation of a parallel system. A parallel algorithm, MatSquare, was developed using the interface the simulation provided. This algorithm meets all the success criteria that was initially outlined, and exceeded these through further optimisations. Additionally, the analysis of the algorithm's execution proved the final success criteria, and went beyond by considering different classes of parallel computation.

5.2 Lessons learnt

Using the proper tools, such as an IDE for writing Java code and `git` for doing revision control, helped immensely with keeping implementation productive. Additionally, creating UML diagrams before writing any code also made managing the large project easier, allowing more time for writing code than trying to figure out how components piece together.

During development, I was often fixated on minor aspects of the parallel simulation interface, which did not end up having much value when writing the parallel MatSquare algorithm. This

caused the interface to have more functionality than required, and this added development time would be better spent on extension work.

5.3 Future work

The input graphs used in the dissertation were very sparse. This could be exploited to reduce the time complexity by using parallel matrix multiplication techniques designed for sparse matrices, such as what Buluç and Gilbert propose [1].

I also only considered one algorithm for solving APSP. It would be interesting to implement others on the simulator and compare how they scale on different types of parallel systems.

Another interesting extension would be to implement the algorithm on real parallel hardware, such as a FPGA, and compare its execution times with the simulated counterpart.

Bibliography

[1] A. Buluç and J. R. Gilbert. Highly parallel sparse matrix-matrix multiplication. *CoRR*, abs/1006.2183, 2010. URL `http://arxiv.org/abs/1006.2183`.

[2] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks. Automatically accelerating non-numerical programs by architecture-compiler co-design. *Commun. ACM*, 60(12):8897, nov 2017. ISSN 0001-0782. doi: 10.1145/3139461. URL `https://doi.org/10.1145/3139461`.

[3] A. Crauser, K. Mehlhorn, U. Meyer, P. Sanders, L. Brim, J. Gruska, and J. Zlatuska. A parallelization of dijkstra's shortest path algorithm. *Proceedings of the 23rd International Symposium on the Mathematical Foundations of Computer Science (MFCS-98), Springer, 722-731 (1998)*, 10 1998. doi: 10.1007/BFb0055823.

[4] J. Dongarra. Report on the sunway taihulight system. Technical Report UT-EECS-16-742, 2016-06 2016. URL `http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf`.

[5] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4(1):17–31, 1987. ISSN 0167-8191. doi: https://doi.org/10.1016/0167-8191(87)90060-3. URL `https://www.sciencedirect.com/science/article/pii/0167819187900603`.

[6] A. Fukunaga, A. Botea, Y. Jinnai, and A. Kishimoto. A survey of parallel A. *CoRR*, abs/1708.05296, 2017. URL `http://arxiv.org/abs/1708.05296`.

[7] V. P. Gergel. Introduction to Parallel Programming, Section 8. parallel methods for matrix multiplication. `http://www.hpcc.unn.ru/mskurs/ENG/DOC/pp08.pdf`, 2005. [Online; accessed 24-April-2022].

[8] D. Howdle. Worldwide broadband speed league 2021, 2021. URL `https://www.cable.co.uk/broadband/speed/worldwide-speed-league/`. [Online; accessed 24-April-2022].

[9] Karp, Alan, and H. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33:539–543, 05 1990. doi: 10.1145/78607.78614.

[10] J. W. Kim, H. Choi, and S.-H. Bae. Efficient parallel all-pairs shortest paths algorithm for complex graph analysis. In *Proceedings of the 47th International Conference on Parallel*

Processing Companion, ICPP '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365239. doi: 10.1145/3229710.3229730. URL https://doi.org/10.1145/3229710.3229730.

[11] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, 1991. ISSN 0743-7315. doi: https://doi.org/10.1016/0743-7315(91)90083-L. URL https://www.sciencedirect.com/science/article/pii/074373159190083L.

[12] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. volume 3633, pages 273–290, 08 2005. ISBN 978-3-540-28127-6. doi: 10.1007/11535331_16.

[13] Sandy Bridge (client) - Microarchitectures - Intel. Sandy bridge (client) - microarchitectures - intel — WikiChip, 2020. URL https://en.wikichip.org/w/index.php?title=intel/microarchitectures/sandy_bridge_(client)&oldid=98305. [Online; accessed 24-April-2022].

[14] P. Sao, H. Lu, R. Kannan, V. Thakkar, R. Vuduc, and T. Potok. Scalable all-pairs shortest paths for huge graphs on multi-gpu clusters. 06 2020. URL https://www.osti.gov/biblio/1814306.

[15] Verizon. IP latency statistics, 2022. URL https://www.verizon.com/business/terms/latency/. [Online; accessed 24-April-2022].

Appendix A

Further definitions

A.1 Abstract algebra

TODO

Appendix B

Other

B.1 Word count

The word count was computed by concatenating the 5 chapters and using TEXcount. I used the following script to do this:

```
echo "$(head -n 4 diss.tex)
    $(tail -n +3 -q sections/*.tex)
    $(tail diss.tex -n +5 -q)" | \
    texcount - $@ -sub=chapter | \
    grep -P "Introduction|Preparation|Implementation|Evaluation|Conclusions" | \
    grep -oP "\d+[+]\d+[+]\d+" | \
    bc | \
    awk '{s+=$1} END {print s}'
```

B.2 Error bars

The plots with error bars:

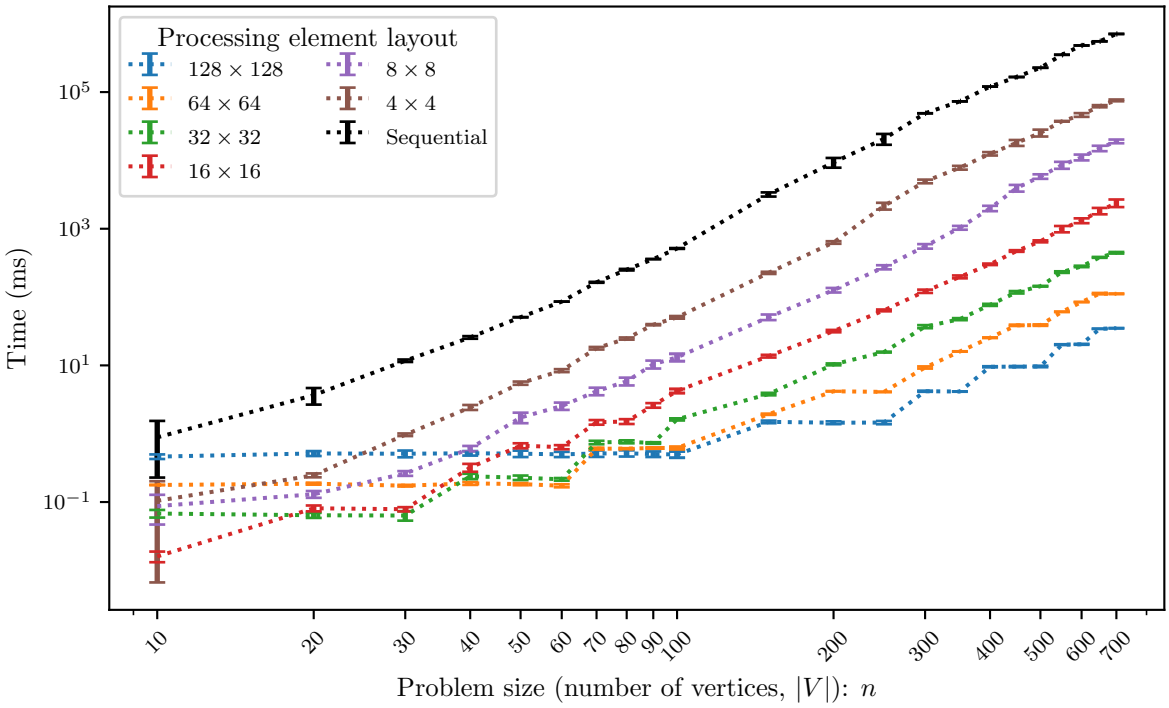


Figure B.1: Sandy bridge total time scaling

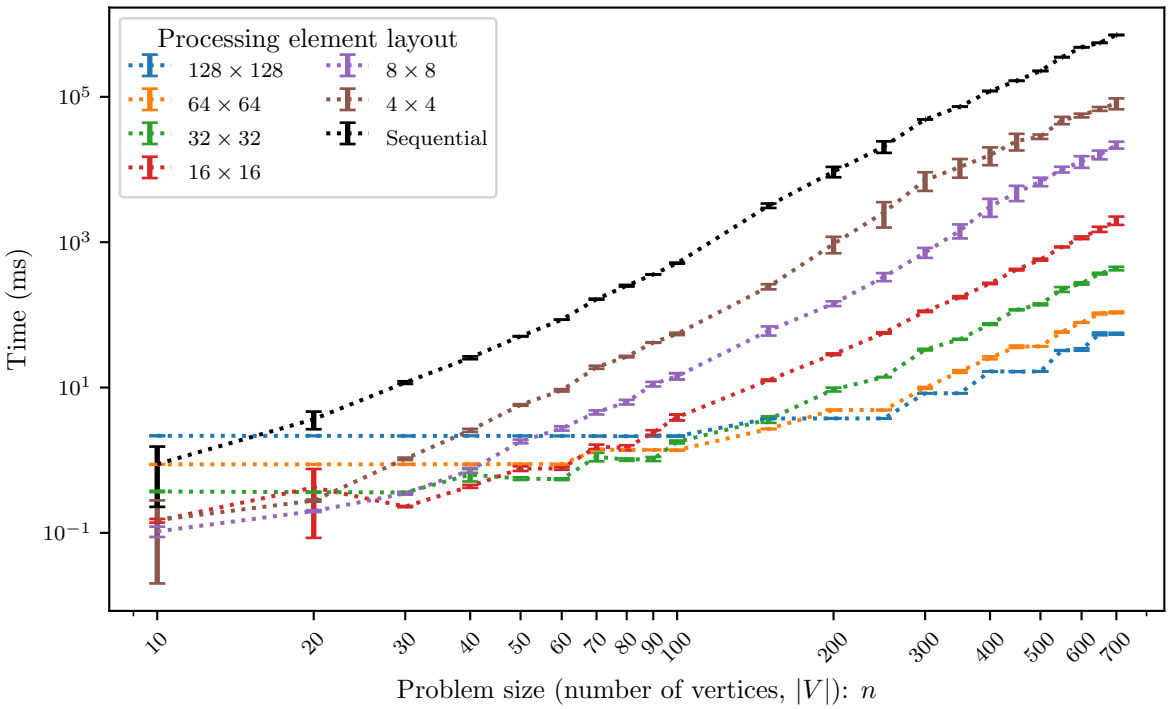


Figure B.2: Taihu-Light total time scaling

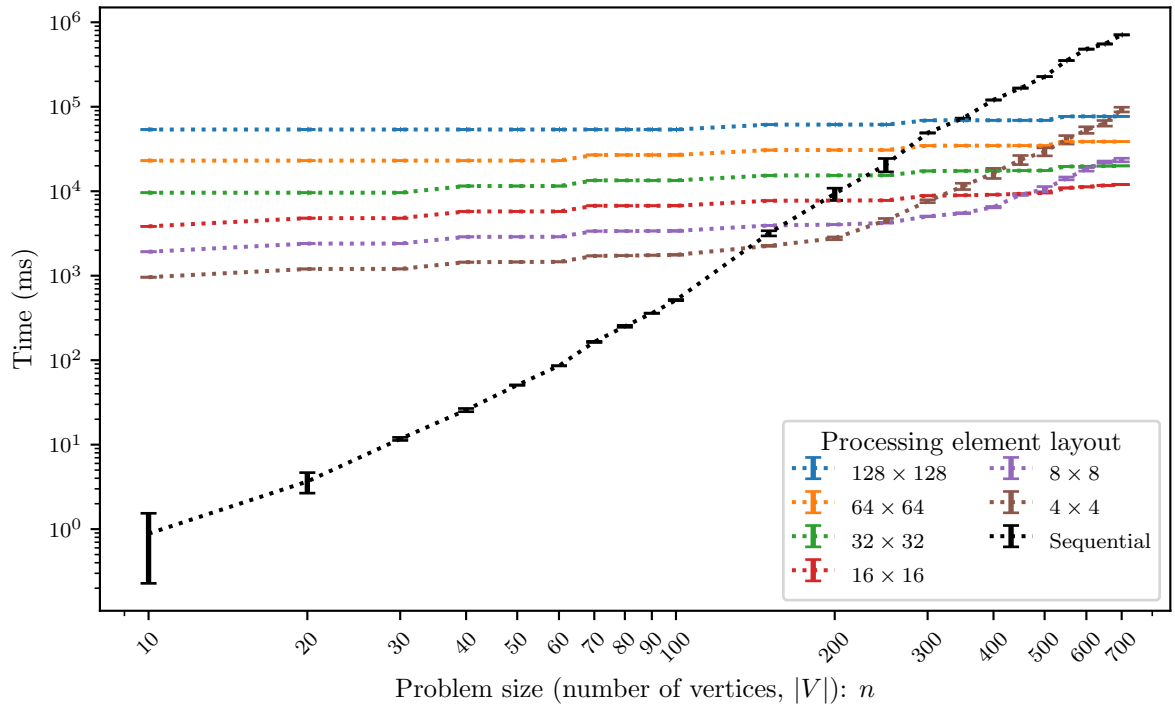


Figure B.3: Internet total time scaling

B.3 Path reconstruction

As an example of the path reconstruction algorithm presented in section 3.3, consider the completed predecessor matrix for the graph in figure 3.7:

$$M_{pred} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 0 & 5 & 0 & 0 & 1 & 2 & 2 & 7 \\ 0 & 1 & 2 & 3 & 1 & 5 & 6 & 7 \\ 0 & 5 & 2 & 3 & 1 & 2 & 2 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 3 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 5 & 2 & 3 & 1 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} \end{array}. \tag{B.1}$$

If we want to reconstruct $0 \rightsquigarrow 1$, we start with the list $[1]$, then prepend $M_{pred}[0,1] = 5$, giving $[5,1]$. We then prepend $M_{pred}[0,5] = 2$. We continue doing this until we get $[0,2,5,1]$, where we stop because $M_{pred}[0,0] = 0$.

Appendix C

Project Proposal

Part II Computer Science Project Proposal

A parallel algorithm for all-pairs shortest paths that minimises data movement

Marcus A. K. September, Clare College

Originator: Dr J. Modi

October 18, 2021

Project supervisor: Dr J. Modi

Director of studies: Prof L. C. Paulson

Project overseers: Prof R. Mantiuk & Prof A. M. Pitts

Introduction

The all-pairs shortest path (APSP) problem is relevant for minimising resource costs when travelling across a road network or a railway network. There are many known serial algorithms for solving this problem, but with a highly parallel processor, there might be more efficient solutions. After multiplying a graph's adjacency matrix with itself n times, it is possible to find the shortest paths of length n between all pairs of nodes. This can be used to construct an algorithm that solves APSP. Matrix multiplication is also a highly parallel problem, meaning there is a lot of potential gain from running it on a multiprocessor.

Many massively parallel processors have hundreds or thousands of CPUs, each with their own private memory. The CPUs then pass data and work to each other using interconnect channels. For problems where the input data is so large that it needs to be distributed across the CPUs private memory, the communication cost is often the performance bottleneck. Therefore, algorithms built for these multiprocessors need to minimize the amount of memory movement. For matrix multiplication, this can be done by employing techniques discovered by G. C. Fox, S. W. Otto, and L. E. Cannon [1, 2].

This project aims to develop an algorithm to solve APSP using matrix multiplication. Since I do not have access to a physical massively parallel processor, it will be simulated in Java. I will then parallelize the matrix multiplication step of the algorithm, and use above mentioned optimizations to minimize the data movement. If time allows, I intend to explore whether the advantage of parallelising APSP is still significant when the problem size is orders of magnitude larger than the number of CPUs in the multiprocessor. The matrix multiplication based algorithm could also be compared with a parallelised implementation of the Floyd-Warshall algorithm, as the two are similar in nature. Further extension work includes optimising the algorithm through graph compression and investigating the effect of different processor interconnect topologies.

Starting point

- I have experience with writing modular, object-oriented code in Java from the Object-Oriented Programming course and the Further Java course.
- I have some knowledge of parallel programming concepts from the Concurrent and Distributed Systems course. I also have a bit of experience using these concepts in code through the Further Java course.
- I know some of the principles behind parallel processors through the Computer Design course

- I am familiar with some route-planning algorithms like Dijkstra's algorithm through the Algorithms course

Resources required

I will use my own computer (an Acer Nitro Laptop), running both Ubuntu Linux 18.04 LTS and Windows 10, to both write and run all of my code. It has a quad-core CPU (an Intel i5-8300H @ 2.30Ghz), 8 GB of RAM, and an NVIDIA GTX 1050 Mobile. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. In case of failure, I plan to use the machines provided by the MCS for development.

The source code, associated data, and my dissertation will be regularly version controlled using a private `git` repository hosted by GitHub. This repository will also regularly be copied over to the Computer Laboratory MCS server.

Substance and structure of the project

The aim of this project is to implement a matrix multiplication based algorithm that solves APSP, and to parallelise the matrix multiplication step to run efficiently on a massively parallel processor with a distributed memory model. That is, the processor will have a large number of processing elements and each of these will have their own private memory. The core part of the project can be divided into several phases:

Data preparation and research The project will start of with a research phase where I look into parallel computing concepts, specifically the importance of minimizing memory movement, and techniques for parallelising matrix multiplication [2, 1]. Research will be done on techniques for evaluating the performance of parallel systems and algorithms, which will be used when planning the evaluation. Additionally, I will plan the different software modules of my project through use of UML diagrams.

The graphs I intend to use will be taken from road network datasets found at [3]. A set of graphs with a wide range of sizes is required for rigorous evaluation of the algorithm. To acquire this, I will create a program that extracts subsections of the datasets. The nodes in the graphs are labelled with their longitude and latitude, so this could for instance be done by removing all the nodes that fall outside of some arbitrarily drawn geographic circle.

APSP algorithm In this step, I will solve APSP using repeated matrix multiplication of the graph's adjacency matrix. The algorithm should also be able to reconstruct the list of nodes that make up the shortest paths. I will start with a serial matrix multiplication routine, and test the algorithm for correctness on small graphs. Later, the matrix multiplication routine will be parallelised and optimisations such as Fox-Otto's will be applied [1].

Simulating multiprocessor and preparing evaluation Since I do not have access to a massively parallel processor, I will simulate one using Java's `Thread` API. Each processing element will be represented by a Java `Thread`, and this will be abstracted away in this phase. Each simulated processing element will have its own private memory, consisting of a few Java variables. The only memory movement will be processing elements sending variables to each other or receiving data from some manager `Thread`. I do not intend to simulate the more complicated aspects of a processor, such as memory mapping, caches and job scheduling.

The framework I develop in this phase will also be used for my quantitative evaluations. One approach is to include a timer in each Java thread and take the maximum of these times after each computation phase. By doing this for all the computation phases of the algorithms – which have communication phases in-between – I would get an estimate of the computation time. For the communication time, I could count the number of memory transfers in the communication phase and estimate how long this would take by using realistic values for memory latency for existing multiprocessors.

Parallelising matrix multiplication In this phase I will implement Fox-Otto's algorithm using the framework developed in the previous phase [1]. After this, the implementation will be integrated into the APSP algorithm.

Possible extensions

1. Generalise the implementation of Fox-Otto's algorithm to also work when the number of processing elements is smaller than the problem size. Then move onto evaluating the advantage of parallelization as this ratio ($\#\ nodes\ /\ \#\ CPUs$) increases.
2. Optimise the APSP algorithm using graph compression techniques, such as removing redundant nodes.
3. Look into different processor network topologies, and how they affect the communication cost of running Fox-Otto's algorithm.

4. Parallelise Floyd-Warshall's algorithm and compare the computation and communication cost with the matrix multiplication-based APSP algorithm.

Success criteria

The project will be considered successful if all of the following criteria are met:

- Implemented an algorithm based on matrix multiplication that can find the length of the shortest path between all pairs of nodes in a graph, and it is able to give the list of nodes that make up such paths.
- Parallelised the matrix multiplication routine of the algorithm to run on a simulated massively parallel processor, where each processing element can send data to each other through simulated interconnects.
- The parallel matrix multiplication routine is optimised to minimise the amount of data movement between processing elements, which is done by using techniques such as Fox-Otto's algorithm [1].
- The evaluation of the algorithm demonstrates that parallel computation gives a high parallel efficiency for solving APSP

Timetable

Week	Date	Description
1 – 3	7 Oct - 27 Oct	**Project proposal and research** • Incorporate feedback on project proposal draft and hand in the final project proposal • Research parallel computing concepts, parallel matrix multiplication algorithms like [1, 2], and how to evaluate parallel systems • Write small fragments of Java code that incorporates these concepts • Set up version control and sort out typesetting of dissertation **Milestones** • Project proposal handed in by October 18th • GitHub repository set up
4 – 5	28 Oct - 10 Nov	**Data preparation and planning** • Create program that extracts subsection of graphs from [3] • Plan evaluation of the algorithm, using knowledge obtained in the research phase • Plan the structure of the code, and how the components interact **Milestones** • Data extractor program written • UML diagrams of core project components created

6 – 8	11 Nov - 1 Dec	*Due to my unit of assessment, productivity during this slot might be lower.* **Simulating multiprocessor I** • Write simulation framework that allows running many CPUs in isolation, measuring their computation time • Write code fragments to test this **Simulating multiprocessor II** • Incorporate an interconnect topology into the framework • Allow the simulated CPUs in the multiprocessor to communicate by sending data to each other through this interconnect network • Create methods to find metrics for the communication cost • Write code fragments to test this **Milestones** • Simple multiprocessor simulation written • Simulation extended to support threads passing each other data • Quantitative computation and communication cost metrics are produced when running test fragments
9 – 11	2 Dec - 22 Dec	**APSP algorithm** • Write serial matrix multiplication routine • Implement algorithm to compute all pairs shortest path using matrix multiplication • Make algorithm reconstruct the list of nodes in the shortest paths • Test algorithm for correctness **Milestones** • APSP algorithm written • Algorithm produces correct results on small graphs
12 – 13	23 Dec - 5 Jan	*This time slot will work as slack time if the project is behind* **Christmas break**

14 – 15	6 Jan - 19 Jan	**Implement efficient parallel matrix multiplication and progress report** • Implement Fox-Otto's algorithm, using the multiprocessor simulation framework • Write progress report **Milestones** • Written parallel matrix multiplication algorithm • Progress report submitted by Friday February 4th
16 – 17	20 Jan - 2 Feb	**Evaluation** • Evaluate the performance of the parallel APSP algorithm, and compare it with a theoretical serial implementation • Compare the advantage of parallelism with theoretical serial implementations as the problem size increases relative to the number of processing elements (extension) **Milestones** • Theoretical comparison between parallel implementation and serial alternative has been made • Quantitative measurement of the performance of the parallel implementation has been done

18 – 20	3 feb - 16 Feb	*This timeslot is dedicated to extension work. If the project is running behind, this slot will instead serve as slack time.* **Extensions** • Generalise Cannon's or Fox-Otto's algorithm to work when the number of CPUs do not match the problem size • Do qualitative evaluation on the algorithm's performance as the ratio # *nodes* / # $CPUs$ increases. • Optimise the algorithm using graph compression • Investigate the performance effect of different interconnect topologies • Parallelise Floyd-Warshall's algorithm and compare its performance with the core algorithm **Milestones** • Theoretical evaluation of the advantage of parallelism as the ratio # *nodes* / # $CPUs$ increases. • Quantitative measurements of the algorithm's performance as this ratio changes have been made • The performance effect of graph compression has been measured quantitatively • A theoretical and/or quantitative evaluation of the effect of different interconnect topologies have been made • Written parallel implementation of Floyd-Warhsall's algorithm
21 – 22	24 Feb - 9 Mar	**Dissertation writing I** • Write first draft of dissertation • Send draft to Director of Studies **Milestones** • Completed first draft of dissertation

23 – 24	10 Mar - 23 Mar	**Dissertation writing II** • Incorporate feedback from first dissertation draft, producing a second draft **Milestones** • Completed second draft of dissertation
25 – 26	24 Mar - 6 Apr	**Dissertation writing III** • Repeatedly review the dissertation • Refine the layout and the diagrams **Milestones** • Submitted final dissertation by May 13th

References

[1] G.C Fox, S.W Otto, and A.J.G Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4(1):17–31, 1987.

[2] H. Gupta and P. Sadayappan. Communication efficient matrix-multiplication on hypercubes. Technical Report 1994-25, Stanford Infolab, 1994.

[3] F. Li. Real datasets for spatial databases: Road networks and points of interest. https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm, 2005.