Marcus A. K. September

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

Computer Science Tripos – Part II

Clare College

April 30, 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: April 30, 2022

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:	**1587**[1]
Final line count :	TODO
Project Originator:	Dr J. Modi
Supervisor:	Dr J. Modi

Original Aims of the Project

To write a demonstration dissertation[2] using LaTeX to save student's time when writing their own dissertations. The dissertation should illustrate how to use the more common LaTeX constructs. It should include pictures and diagrams to show how these can be incorporated into the dissertation. It should contain the entire LaTeX source of the dissertation and the makefile. It should explain how to construct an MSDOS disk of the dissertation in Postscript format that can be used by the book shop for printing, and, finally, it should have the prescribed layout and format of a diploma dissertation.

[1] This word count was computed by `echo "$(head -n 4 diss.tex) $(tail -n +3 -q \ sections/*.tex) $(tail diss.tex -n +5 -q)" | texcount -total` i.e. by concatenating the subfiles and using TeXcount.

[2] A normal footnote without the complication of being in a table.

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

None

Contents

List of Figures

List of Tables

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

Acknowledgements

This document owes much to an earlier version written by Simon Moore Moore (1995). His help, encouragement and advice was greatly appreciated.

Chapter 1

Introduction

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 transitory 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 element (PE). Today, there are a huge variety of different parallel systems.

Designing algorithms to explore the computational resources of parallel systems is important. To fully utilise the computational power of these new systems to increase performance, a new kind of algorithms must be considered. Given a serial algorithm that solves a problem, there is a limit to how much performance increase automatic parallelisation can achieve. 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 higher 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 a source vertex i to a sink vertex j. In the *All-Pairs Shortest Paths* (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, route packets across the Internet to maximise some given metric, or planning 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 Fukunaga et al. (2017); Crauser et al. (1998); Sao et al. (2020); Kumar and Singh (1991); Kim et al. (2018). 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 Kumar and Singh (1991); Crauser et al. (1998). Kim et al. and Sao et al. have analysed the performance of the algorithms through experimental measures, but they did not run them on more than 256 PEs. Additionally, their test graphs very 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,

Chapter 2

Preparation

Introduction goes here.

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 n^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 Dongarra (2016). 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 (Karp et al., 1990). 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.

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

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

well as discussing alternative approaches to parallelising APSP, we can better understand the 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.

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, 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, by replacing 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) gives 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

[2] See appendix section A.1.

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

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.
- 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 into our inner-loop.
- If simulated, this model gives the programmer more flexibility...

Message passing The PEs are interconnected through point-to-point links, arranged in some fully-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

TODO: Mention networkX in python for graph generation

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.

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 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.

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

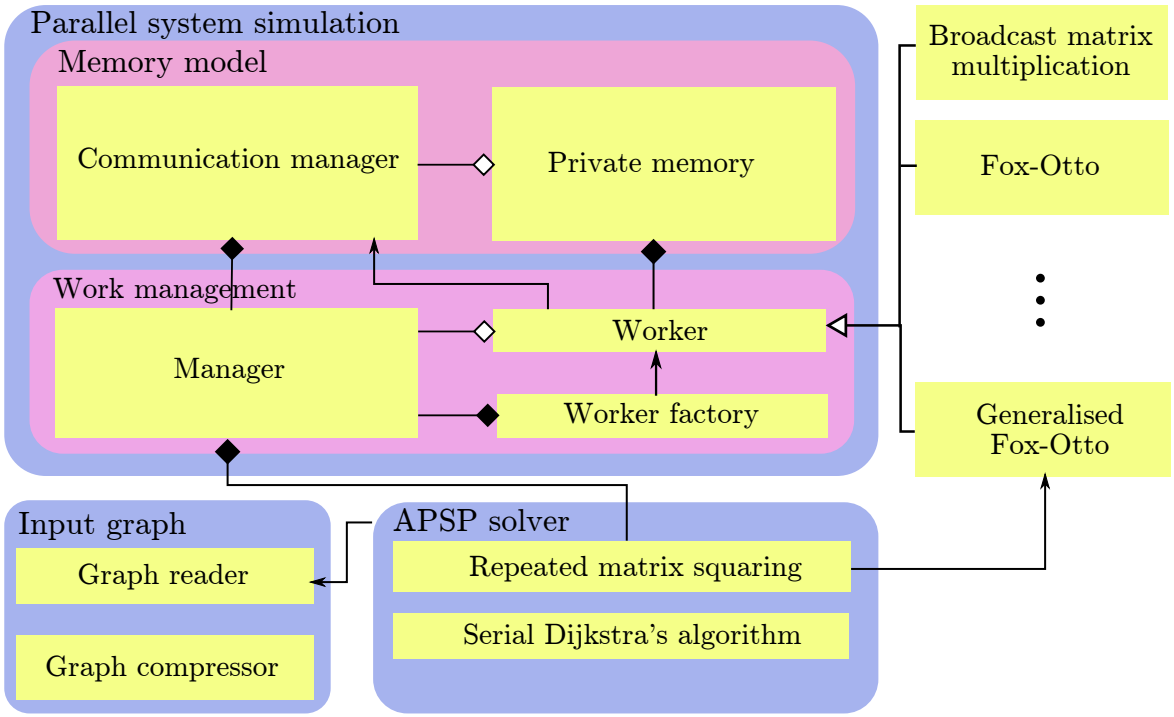


Figure 2.1: High-level overview of the implementation

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.

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 Conclusion

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

Chapter 3

Implementation

Some kind of introduction, referring to the code repository goes here...

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 used in (Li et al., 2005) and has been made available on the author's website[1]. This graph was used to prove that practical problems can quickly be solved on my simulated parallel system, as long as graph compression is used. However, for evaluation I needed graphs of various sizes, so these were generated randomly.

3.1.1 Random graph generation

I used Erdös-Rényi graphs, specifically the $G(n, p)$ model. In this model, a random graph is generated by starting with a graph of n vertices with no edges. We then consider independently include each of the n^2 edges with probability p. This model has previously been used to evaluate the performance of APSP algorithms Crauser et al. (1998). Additionally, it allows specifying the graph size, so we can measure the execution time of the algorithm for a wide range of evenly spaced inputs.

Choice of parameter p

The random graphs are not real-world graphs, but we can set the parameter p such that the Erdös-Rényi graphs are as similar as possible to real-world road networks. This will

[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.

make the evaluation results more applicable. There are many characteristics of a graph, such as vertex- and edge-connectivity, betweenness centrality, clustering coefficient, and average distances. However, trying to generate new graphs with similar values for all of these metrics quickly becomes a difficult problem. 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 fully-connected, as most road-networks are. In the $G(n, p)$ random-graph model, the number of edges follow 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*[2], and including each remaining edge independently with

[2]Each vertex has exactly 2 neighbours.

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 for various road-network datasets[4].

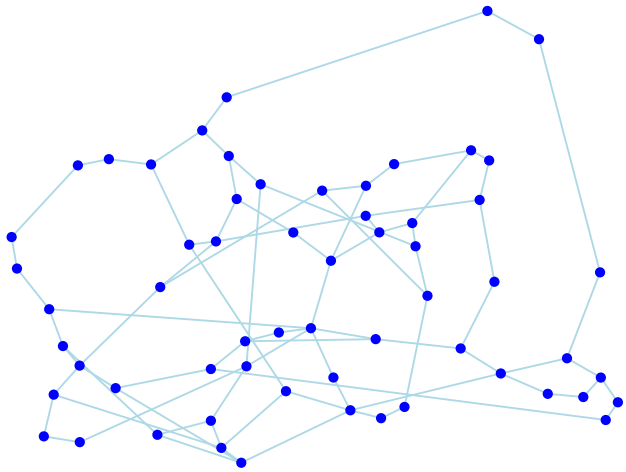


Figure 3.2: A visualisation of a graph generated with the modified Erdös-Renyi model. The graph has $n = 60$ vertices, and the length of the edges correspond to their weight[6].

probability

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

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

I chose the desired average degree based on the Californian road-network, which is 2.061. 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, from 10 nodes to 700 nodes for use in evaluation[3].

[3]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`

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

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

3.2 Simulation of a distributed memory multiprocessor

The parallel system simulation sprouts out from the `Worker` interface it provides. The parallel programmer can use this interface to implement a parallel algorithm by describing how each processing element (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 sent to the `Manager` which creates p^2 `Worker`s using a factory, and the parallel programmer just needs to call `Manager::doWork()` for all the workers to start executing. The `Manager` will then efficiently interleave the execution of the worker phases, and use 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, which also takes stalling causes by MIMD execution into account. This timing analysis functionality is implemented by using the decorator pattern.

3.2.1 Overview of components

In Figure 3.3, we see an overview of the main components of the simulator, and how they interact. The `Manager` constructs the specified number of `Worker`s based on some provided subtype e.g. `FoxOtto`, using a `WorkerFactory`. When `doWork()` is called, it will then run the `Worker`s 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 algorithms ....

3.2.2 Work management

When constructing the `Manager`, we specify the number of PEs, their initial memory content, and what computation they should do in each phase, which is done by specifying the class of some implementation of *`Worker`*. When calling `doWork()` on the manager, it will then run all the workers through a specified number of phases, where there is 3 subphases in each phase. Looking at a single worker in isolation, the execution can be seen in algorithm 1. However, these tasks are simulated in parallel, and we must run the same subphase of all the workers before moving onto the next because of data dependencies. This parallel execution order is visualised in Figure 3.4. After this is done, we can call `getResult(label)` to extract the result by combing the private memory of each worker into a matrix.

The worker done by each PE is specified in subphases. When investigating the different algorithms, as well as possible extension like Floyd-Warshall, I realised that they all had clear distinctions between when the PEs were doing computation and when they were communicating. This made forcing such a distinction in the interface not induce a reduction in flexibility. Instead of providing an abstract method `work`, where the programmer would themself create a `for`-loop and combine communication and computation, we provide the four methods shown in Figure 3.3 and execute them as shown in algorithm 1.

Algorithm 1: Execution of PE(i, j)

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

Splitting up the work phases allows them to be represented as independent `Callable` tasks, which has many benefits. Firstly, it allows managing the workers using higher-level constructs like a Java `ExecutorService` rather than using Java `Thread`. We can submit all the p^2 tasks in a subphase to the executor service, and it will complete all of them without ever spawning more than 16[7] threads, which avoids a lot of overhead. Additionally, as I will explain further in the memory model, the side-effects of communication only happen when `CommunicationManager::flush` is called, so there are no data dependencies between PEs within a subphase, only between two different subphases. Therefore I do not need locking mechanisms inside the ***`Worker`***, but can instead implicitly implement them by executing them in the order shown in Figure 3.4. The third benefit is allowing repeated execution of the same work, which gives a more accurate timing estimate.

The `WorkerFactory` uses Java's reflection API to create instances of any implementation of the ***`Worker`*** interface. When constructing this factory, by provide a `Class` object of for instance the `FoxOtto` implementation. It can then infer the appropriate constructor at runtime and create p^2 `FoxOtto`-workers, without there being need for a dedicated Fox-Otto worker factory. This simplifies the parallel programming interface as implementing ***`Worker`*** is all that is needed to add an additional parallel algorithm to the system.

3.2.3 Memory model

Memory is accessed through string labels. The communication Each PE has its own local memory, represented by a `PrivateMemory` object. To make access as easy to use as possible for the programmer, access is done through string labels. For example, a

[7] This number is configurable, but since my Laptop can execute 16 threads simultaneously, this was the optimal number

PE can store some values at location `"A"` and then retrieve it with the same label later, additionally, for PEs handling a non 1×1-sub-matrix, the local memory is arranged in a matrix where the PE can for instance store values at location `(1,4,"A")` to associate it with some result related to for example $C_{5,9}$.

Introduction. To send for example data with point-to-point connections, both the sender and recipient must call `sendData` and `receiveData`, respectively. The `receiveData(i, j, label)` method does not return a value, but instead causes the recipient's memory at label `label` to be set to whatever value was received. This change does not happen until `CommunicationManager::flush()` is called, which happens in the synchronisation phase in ?.During the flushing all the sent data is matched up with corresponding calls to `receiveData`, and the private memories of all the recipient is modified by the `CommunicationManager`. An obvious alternative to this approach is making the `receiveData` method return the actual number received, which can be done by putting the recipient Thread to sleep until another thread has sent it data. My approach has the following benefits and drawbacks:

+ Separating the computation and communication gives implicit thread synchronisation, which allows using fewer `Threads`, causing less overhead (see)
+ Measuring the computation time for evaluation is a lot simpler
− Computation must be interleaved with a communication phase and we get less flexibility for algorithms where these two phases cannot be cleanly seperated
− The `receiveData` method not returning anything is not an intuitive abstractions, but rather requires the programmer to know slightly more about the implementation of the `CommunicationManager`

The main benefits are making the work management simpler, and setting up for more accurate evaluation. The first drawback is not relevant for the algorithms I have used.

3.2.4 Estimating computation and communication times

* Diagram of how communication causes stalls, counted as comTime: * Split up phases into the 3 things, and can use arrow to indicate which PE is sending to what, and assumption on if send-before, no delay to start next phase... * What is measured as computation time * The equation for estimating communication, latency + bandwitdh * w, and the assumptions made behind this

TODO: Cite this Gergel (2005), for details on $\alpha + w \cdot (n^2/p^2) \cdot \beta$

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 a certain subphase took to compute, stopping the clock whenever we enter a communication phase. We also want to minimize the effect of running this in a simulated environment, compared to on a real parallel system, where we would have dedicated private memory for each PE. Possible effects might include the JVM suddenly performing garbage collection or the PE's private memory is not in cache. To combat this, we want to repeat each phase several times and take their average 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 used in other work that ... 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

$$t(i, j, m) \triangleq \delta(i, j) \cdot (\alpha + \beta \cdot \text{size}(m)). \tag{3.3}$$

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. 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. 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

$$T_i^{(n+1)} = \max \left(\overbrace{T_i^{(n)} + \sum_{(j,m) \in R} t(i, j, m)}^{\text{All necessary data has already arrived}} \quad , \quad \underbrace{\max_{(j,m) \in S} \left(T_j^{(n)} + t(j, i, m) \right)}_{\text{PE } i \text{ must stall until data has arrived}} \right). \tag{3.4}$$

This start time and how it is affected by late senders can be visualised in figure

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. In Figure 3.5, we see an overview of the components of the `timingAnalysis` package, where the green classes are implemented using decoration: They extend the base class and only override methods related to timing, and use a reference to the base object to perform the original functionality in addition to the added timing analysis.

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 broadcasting. We also implement the δ function using the `Topology` interface. Together, these two classes are used to implement the function in Equation 3.3, used whenever we compute the time it takes to send a message in `TimedCommunicationManager`.

The main timing simulation functionality happens in `TimedCommunicationManager`. Here, we maintain a matrix of the simulated times of each PE, starting with $T^{(0)}_{(i,j)}$ for each of the p^2 PEs. We then update these times whenever we advance from phase (n) to phase $(n+1)$, with different rules depending on whether we have a communication phase or a computation phase. For computation phases, we simply increment the time with the measurement from the `TimedWorker`. For the communication phases, we use the formula in Equation 3.4, where we build up the sets S and R through decoration of the methods `sendData`, `broadcastRow`, etc. We also only evaluate this formula when executing the decorated `flush()` method at the end of each communication phase. We also have separate counters for how much of this simulated execution has been used for communication, counting both time to send data and stalling until data is ready.

Algorithm 2: Measuring computation time in `TimedWorker`

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

```
/* Enable read-only       */
1 computation(l);
2 t_0 ← current thread CPU time ;
3 for i = 0 ... num. repeats − 1 do
4 |   computation(l);
5 end
6 t ← (current thread CPU time − t_0) / num. repeats
/* Disable read-only      */
7 computation(l);
```

In `TimedWorker`, we use a clever trick to get more accurate measures for the computation time, as shown in 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 reduce the effect of short-time execution differences on the host system.

There are several benefits to separating the timing functionality from the main functionality of the work management and memory model components, as I have done. Firstly, it makes the program more modular, which makes implementation easier as I could focus on fewer things at a time. It also allows the main components to be further decorated with other timing behaviour, for example to simulate SIMD without modifying existing code. In Figure 3.5, it looks like I am creating a whole new layer of classes and interactions, but most of these are already implemented in the basae components, and the interactions do not need to be reimplemented in the decorators.

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, then we look at how the distance product used to compute lengths of shortest paths in the graph, just starting from the graph's adjacency matrix. We then look at how the predecessor matrix is incrementally computed, and how it is used to reconstruct the paths. When describing the how we find the distance- and predecessor matrix, we will look aside from how the matrix multiplication procedure is implemented, only focusing on its desired semantics. Afterwards, we describe how the matrix-multiplication computation is distributed over the PEs, emphasising how the algorithm each PE executes satisfies the desired semantics assumed of the procedure. Throughout the description of MatSquare, I will refer to the example graph in Figure 3.6 to better illustrate how it works.

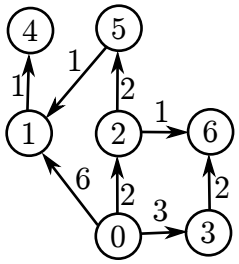


Figure 3.6: An example of a directed graph with 7 vertices.

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,

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
```

we will let n be the number of vertices in this padded graph, and the adjacency matrix A will also include inserted empty nodes.

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 $i \rightsquigarrow j$.*

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[8].

The MatSquare algorithm

The goal of the MatSquare algorithm is to compute 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. Using the adjacency matrix A for the example graph in Figure 3.6, $A^4_{0,1}$ would be the length of for instance 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, one way to compute the distance matrix is

$$M_{dist} = M^{(n)}_{dist} = \underbrace{M^{(1)}_{dist} \otimes \cdots \otimes M^{(1)}_{dist}}_{n-1 \text{ times}} = \underbrace{A \otimes \cdots \otimes A}_{n-1 \text{ times}} \tag{3.5}$$

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 $M^{(n')}_{dist}$ 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 $M_{i,j} \leftarrow \min_{0 \leq k < n}(M_{i,k} + M_{k,j})$ 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

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

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*

$$w_{i,j} = \operatorname*{argmin}_{0 \le k < n}(M_{i,k} + M'_{k,j}). \tag{3.6}$$

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 loop. This will obstruct the path reconstruction algorithm, so we want to keep the predecessor from path $i \rightsquigarrow k = j$ in those cases.

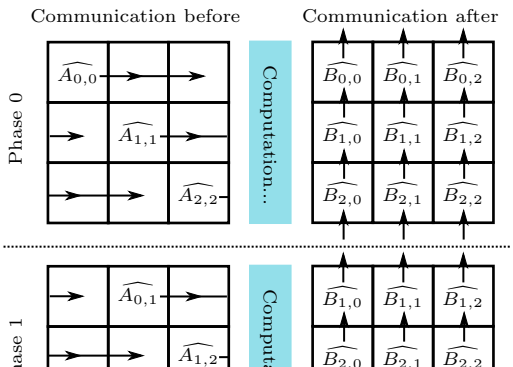
The algorithm to reconstruct the paths is simple once we have computed the predecessor matrix M_{pred}. Given a query $i \rightsquigarrow j$, we start with a list of nodes $[j]$, and repeatedly prepend the predecessor of the first node. We stop when find the predecessor of a node points to itself. This is also why we initialise the predecessor entries $M^{(x)}_{pred}[i, j]$ to j if there are no paths $i \rightsquigarrow j$ of length x. As an example, the completed predecessor matrix for the example graph in Figure 3.6 is

$$M_{pred} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 0 & 5 & 0 & 0 & 1 & 2 & 2 & 7 \\ 0 & 1 & 2 & 3 & 1 & 5 & 6 & 7 \\ 0 & 5 & 2 & 3 & 1 & 2 & 2 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 3 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 5 & 2 & 3 & 1 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} \end{array}, \tag{3.7}$$

so 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 $M_{pred}[0, 0] = 0$.

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 line 6 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}. After the parallel procedure, these three matrices should be updated according to lines 7–13 in algorithm 3.

To parallelise the procedure, we distribute the data 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, PE with id $(0, 0)$ in ... would have .... $A, B, P[0 \ldots 1, 2 \ldots 3]$. It would also be responsible for computing the submatrix of the distance product $A \otimes B$ and the next predecessor matrix with the corresponding entries, that is $M_{dist}[0 \ldots 1, 2 \ldots 3]$. We will abbreviate the submatrix size $\lceil n/p \rceil$ as n'. Additionally, we will refer to A', B' and P' as the submatrices each PE currently holds, but that are shifted around in communication phases. Each PE also holds an additional copy of the initial A', called A'_{const} that is not moved around.

If we consider the computation of a single entry in the distance product as in Equation 2.4, we notice that the order of products $A_{i',j} \cdot B_{k,j'}$ that we consider 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 minimize over all the $0 \leq k < n$ simultaneously. Fox et al. describe a data movement technique for doing this Fox et al. (1987). As seen in Figure 3.7, this technique consists of broadcasting the submatrices of A along the rows before partially computing the min. After computation, the submatrices of B is shifted upwards to the next PEs. This pattern ensures that in the computation phases, all the PEs have access to matching submatrices of A and B in the distance product. These communication phases are implemented in lines 8–10 and 25–26 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 11 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 12 and compare the new distance term with the current minimum distance on lines 16–17 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 3.7. In fact, it can be shown that on line 16, we are computing $A[i', k] + B[k, j']$ with i', j' and k as defined on lines 13–15. Therefore, the witness will be k, which makes the predecessor update logic on lines 19–21 very similar to what is done in lines 10–12 in algorithm 3.

3.4 Graph compression

Most road networks tend to be very sparse, so many of the vertices will just have two neighbours. We can leverage this to reduce the amount of computation as there is only one path across a segment of two-degree nodes. We do this by contracting all the contiguous paths of two-degree nodes into a single edge. In Table 3.2, we see that this reduces the problem size by over 16% for all our datasets, and for some of them, like the California road network, we have massive gains.

We compress the graph by removing all nodes with exactly two neighbours, and the corresponding 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 3.8. 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 set the other junction to be one of the two-degree nodes that neighbour the junction. This is done to not introduce further edge-cases in the path-reconstruction later. Second, 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 to queries on 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 $p \rightsquigarrow q$ on the original graph, we visualise the graph as shown in Figure 3.9. We have 5

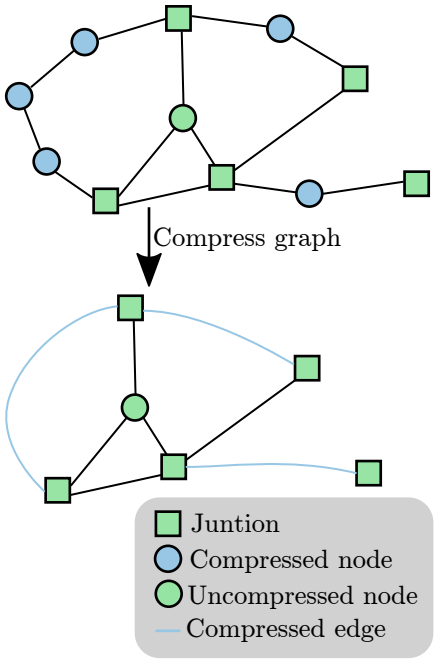


Figure 3.8: Graph before and af

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

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

`// ** Initialisation phase **`
1 $M'_{dist} \leftarrow n' \times n'$ matrix;
2 $M'_{pred} \leftarrow n' \times n'$ matrix;
3 **for** $0 \leq i_2, j_2 < n'$ **do**
4 $\quad M'_{pred}[i_2, j_2] \leftarrow j$;
5 $\quad M'_{dist}[i_2, j_2] \leftarrow \infty$;
6 **end**
7 **for** *phase number* $l = 0$ *to* $p - 1$ **do**
$\quad$ `// ** CommunicationBefore phase` l `**`
8 $\quad$ **if** $j = (i + l) \mod p$ **then**
9 $\quad\quad$ `broadcastRow(`A'_{const}`)`;
10 $\quad$ **end**
$\quad$ `// ** Computation phase` l `**`
11 $\quad$ **for** $0 \leq i_2, j_2 < n'$ **do**
$\quad\quad$ `// We start with` m `s.t.` $k = i'$ `at` $l = 0$`, which causes shorter`
$\quad\quad$ `// paths from the previous squaring to be considered first.`
$\quad\quad$ `// This is necessary to get the predecessor right.`
12 $\quad\quad$ **for** $m = i_2, i_2 + 1, \ldots n' - 1, 0, 1, \ldots, i_2 - 1$ **do**
$\quad\quad\quad$ `// In this iteration, we compute A[i', k] + B[k, j'], where`
13 $\quad\quad\quad k \leftarrow (n' \cdot (i + l) + m) \mod n$;
14 $\quad\quad\quad i' \leftarrow i \cdot n' + i_2$;
15 $\quad\quad\quad j' \leftarrow j \cdot n' + j_2$;
$\quad\quad\quad$ `// Distance product`
16 $\quad\quad\quad d_{new} \leftarrow A'[i_2, m] + B'[m, j_2]$;
17 $\quad\quad\quad$ **if** $d_{new} < M'_{dist}[i_2, j_2]$ **then**
18 $\quad\quad\quad\quad M'_{dist}[i_2, j_2] \leftarrow d_{new}$;
$\quad\quad\quad\quad$ `// If` $k = j'$`, the path` $k \to j'$ `will be a self-loop, so`
$\quad\quad\quad\quad$ `// we should not update the predecessor`
19 $\quad\quad\quad\quad$ **if** $k \neq j'$ **then**
$\quad\quad\quad\quad\quad$ `// The cell` $P'[m, j_2]$ `holds` $P_{k,j'}$
20 $\quad\quad\quad\quad\quad M'_{pred}[i_2, j_2] \leftarrow P'[m, j_2]$;
21 $\quad\quad\quad\quad$ **end**
22 $\quad\quad\quad$ **end**
23 $\quad\quad$ **end**
24 $\quad$ **end**
$\quad$ `// ** ComunicationAfter phase` l `**`
25 $\quad$ `send(`$p + i - 1 \mod p$`,` j`,` B'`)`;
26 $\quad$ `send(`$p + i - 1 \mod p$`,` j`,` P'`)`;
27 **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

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, and we just compute the path length from the bookkept information (This case is not depicted on the figure).
2. We have $p = E, q = F$. We compute the four possible paths, $p \rightsquigarrow \{A, B\} \rightsquigarrow \{C, D\} \rightsquigarrow q$, and use the shortest one.
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:

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

where $p = J_1, J_1 = J_2$, etc. are possible. To map this back to a path in the original graph, we must expand all the edges in the path between the junctions, if present, $J_1 \rightsquigarrow J_2$ to a path in the original graph. This is done by going through each edge, and swapping the compression edges out for the list of edges that were compressed.

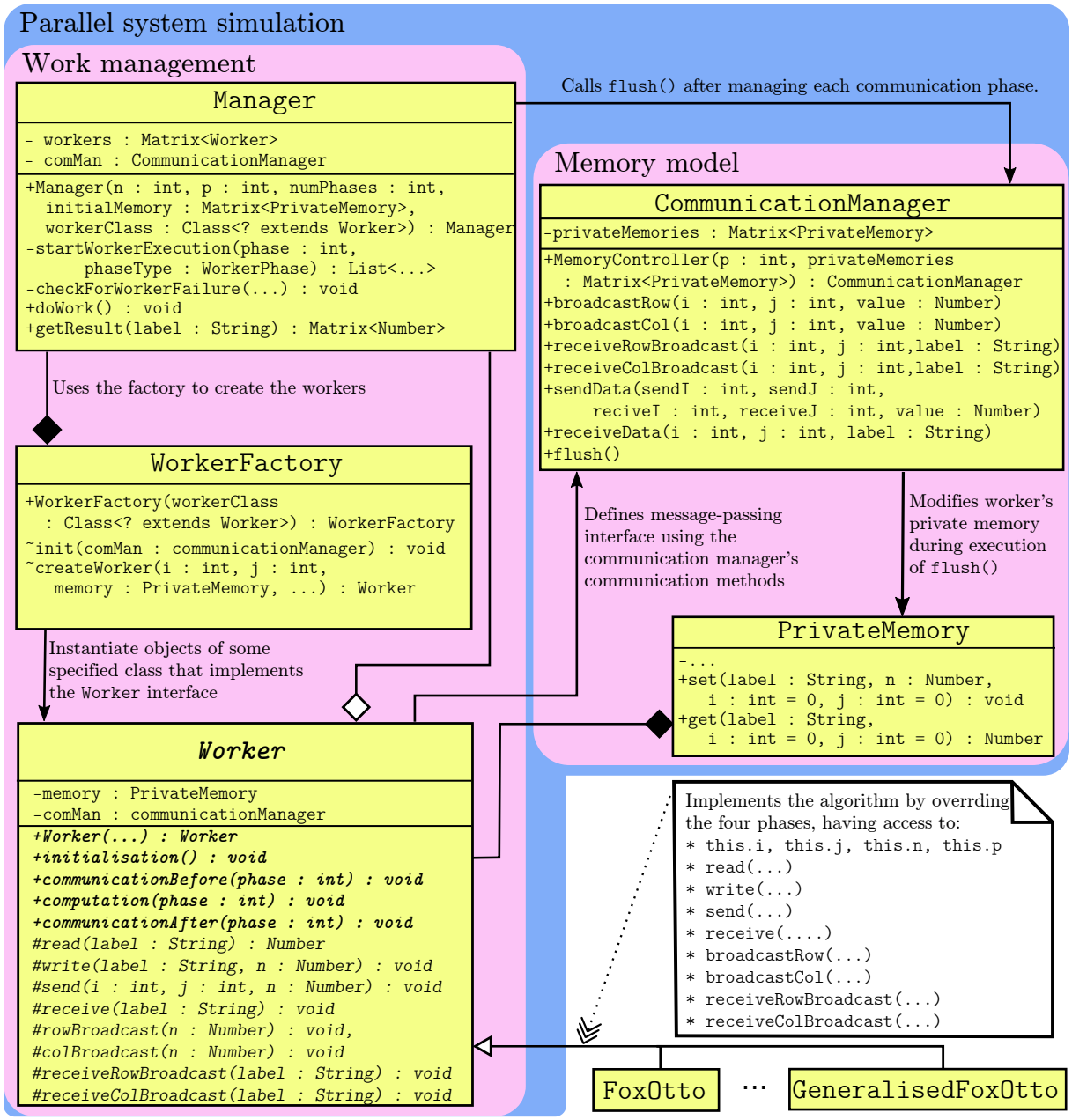


Figure 3.3: A overview of the classes and their interaction in simulator.

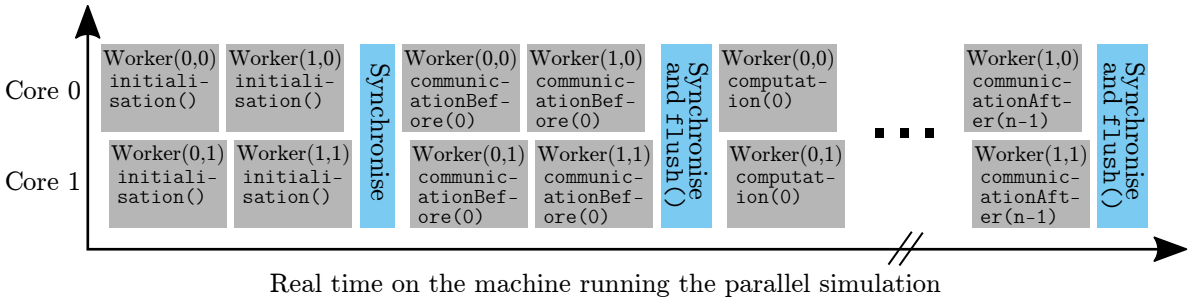


Figure 3.4: Example of multithreading when simulating a 4 processing elements on a host computer with 2 cores.

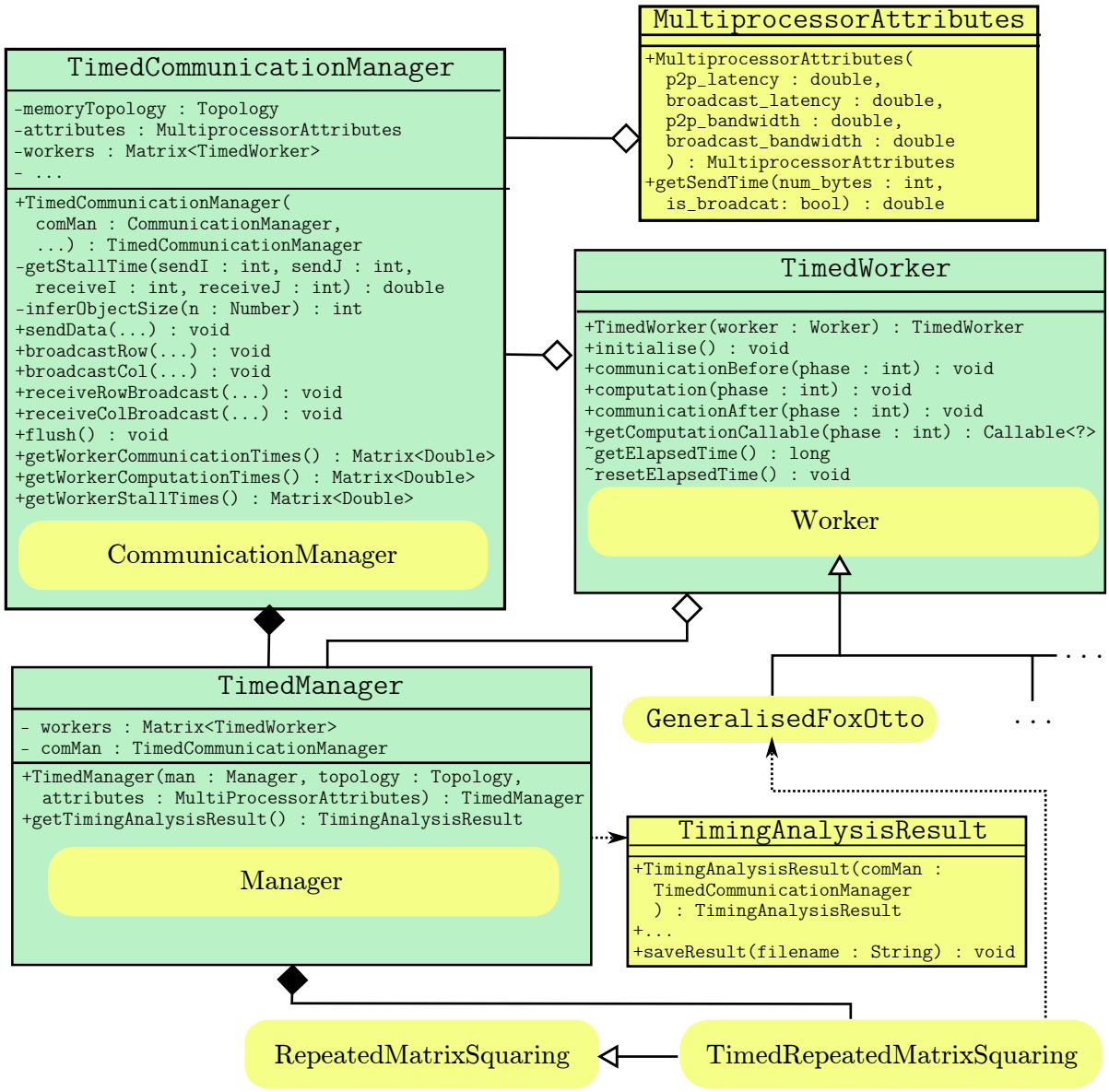


Figure 3.5: A overview of the components of the `timingAnalysis` package. Note that the `GeneralisedFoxOtto` and the `RepeatedMatrixSquaring` classes are not part of this package. Additionally, the green class diagrams with a yellow class inside it are shorthands for the decorator-pattern, where the outer class extends the yellow inner one and contains a reference to it.

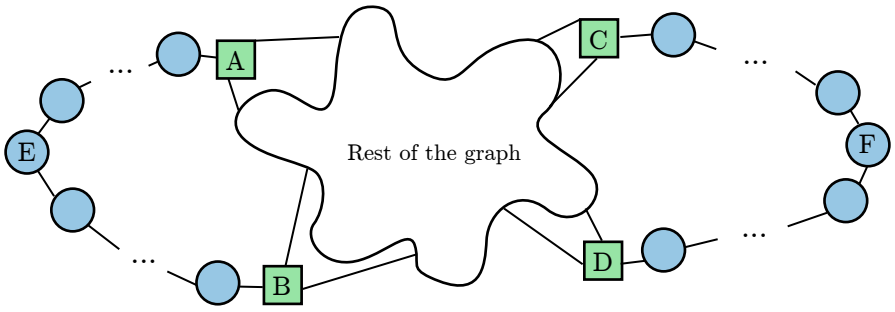


Figure 3.9: A graph

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 and number of PEs.

4.1 Parallel MatSquare

Introduction. I successfully implemented the MatSquare algorithm. To test the correctness, I set up several unit tests using small graphs where I manually set up the model answer. See ... 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 3. To assert the correctness of the path lengths and the path reconstruction, I manually computed the results for

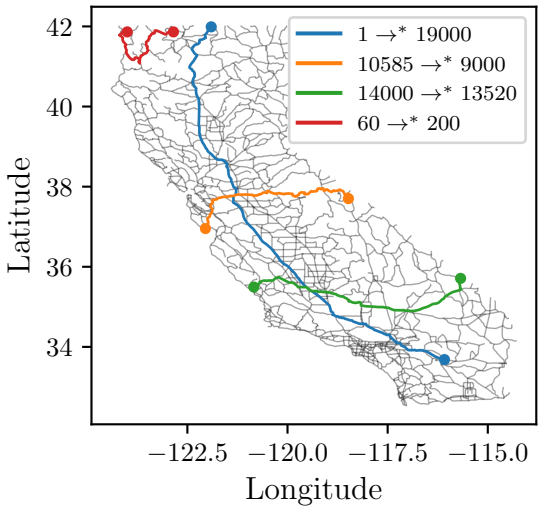


Figure 4.1: Four example shortest-paths in the Californian road network ($|V|$= 21048, $|E|$= 21693). These paths consist of 575, 328, 272, and 207 nodes, respectively.

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 $|V|^2$ pairs. To do this, I implemented Dijkstra's algorithm with path reconstruction, which is a lot 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 vertex-pairs. This was done in a unit test for reproducibility, and was 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 also work. I tested this by running the MatSquare algorithm on the compressed graph and comparing the distances found as well as the list of vertices making up the paths, with an APSP algorithm running on the uncompressed graph. This gave the same results for all $|V|^2$ pairs on a large graph with more than 400 nodes. 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 also be infeasible to do in a short amount of time were it not for the graph compression optimisation.

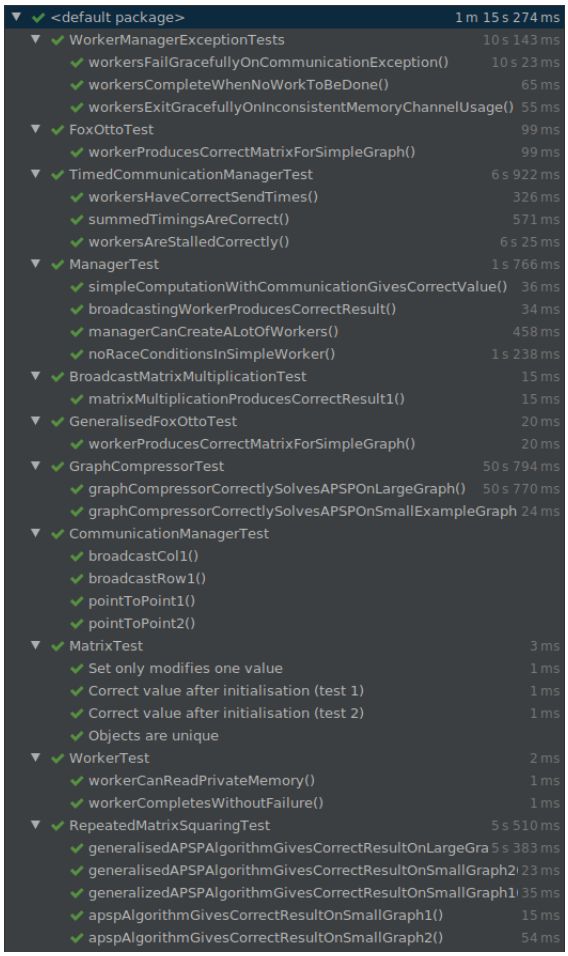


Figure 4.2: A screenshot showing all the unit tests written and that they passed.

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

[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.

technique Fox et al. used. As we see in 4, this makes all the PEs have exactly the data that they need at each computation phase, after just sending a packet across a single interconnect channel, a point-to-point link and the row-broadcast highways. 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 the parallel programmer is very simple, but it had 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. This programmer interface has been thoroughly tested through unit tests that cover usage much more complex than the what was needed in algorithm 4. The computation time measures are done with uncertainty in mind to reduce the amount of noise in the measurements. Additionally, a MIMD model is used for the communication time, which is more complicated to implement compared to a naïive SIMD model, but with the benefit of giving more realistic measures. Lastly, the simulation runs in parallel itself by simultaneously executing work subphases. 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 was close to 800% while doing so, showing the parallisation was successfull.

I also successfully "*[p]arallelised the matrix multiplication routine of the [MatSquare] algorithm to run on*" the parallel system simulation. This is done through my implementation of the `FoxOtto` and `GeneralisedFoxOtto` 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

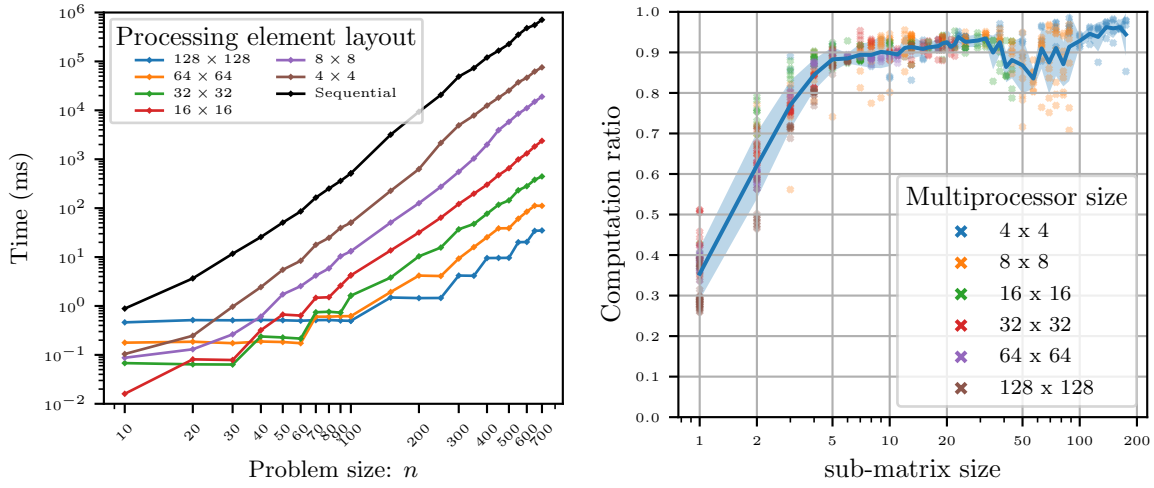
TODO

4.3.1 Choice of communication parameters

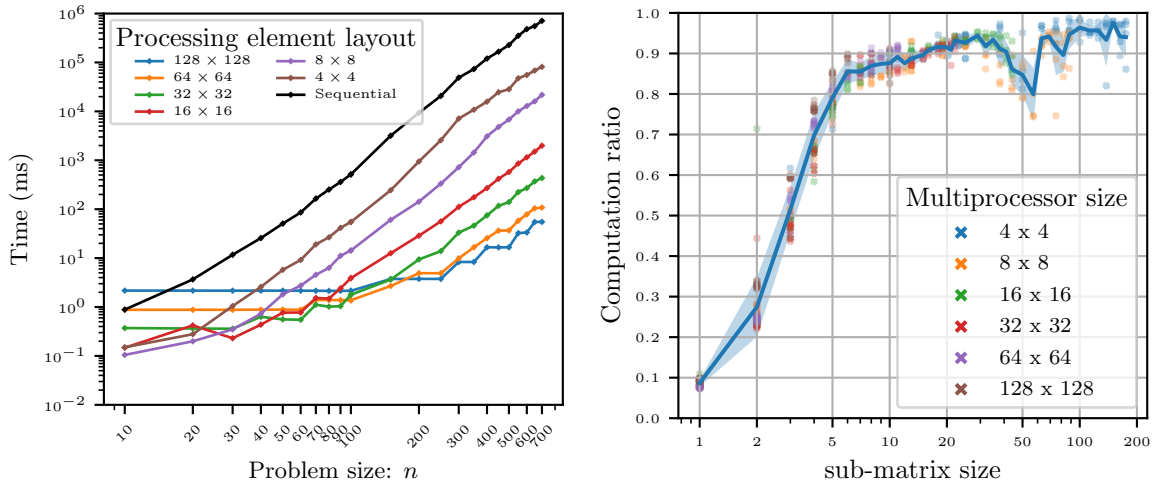
By basing the communication constants on real hardware, we can draw more applicable conclusions. For the parallel system simulation to estimate the message-passing delay, we must specify a latency and bandwidth. As we reviewed in section 2.1, there are many classes of parallel computers, ranging from multi-core processors to different computers communicating over the Internet. To get a sense of what the parallelism benefit is for this whole range, we consider the two extremes and a supercomputer in-between. We consider three real-life systems within each of these three classes, and use their communication constants. 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 Sandy Bridge (client) - Microarchitectures - Intel (2020). For one core to send a 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 it must then wait for the cache line to come back. Campanoni et al. have done measurements of this kind of cache coherence latency, and found that for the Sandy Bridge microarchitecture, the round-trip latency is about 107 80% of the time Campanoni et al. (2017). I make the simplifying assumption that the latency is constantly at 107 cycles, and this delay is fixed whatever the clock frequency is, as I will be using the clock frequency of my own laptop. In the ring interconnect used to implement cache coherency in the Sandy Bridge microarchitecture, the data ring used has a bandwidth of 32 bytes per cycle. I will therefore make the assumption that when sending data from one private L1 cache to another through the ring interconnect (using cache cohernccy protocol), this is the available bandwidth.

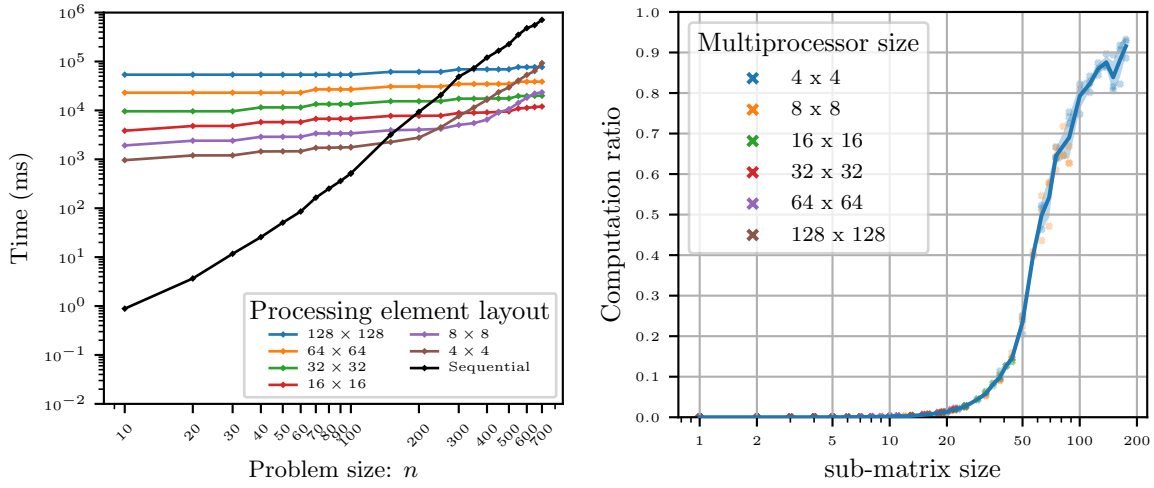
Supercomputer For the supercomputer parallel system, I have based by constants on the Sunway TaihuLight system, which consists of several SW26010 processors connected together through a system interface with a bidirectional bandwidth of 16 GB/s and a latency around 1 µs Dongarra (2016). Each SW26010 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 decided to not simulate this memory hierarchy between the PE, but instead assume each individual



(a) Multi-core processor (Sandy Bridge)



(b) Supercomputer (Sunway TaihuLight)



(c) Distributed computer (Internet)

Figure 4.3: In the left column, the total execution time of MatSquare is plotted for different different input graphs with $n = |V|$ vertices. In the right column, computation ratios are grouped by the number of rows and columns in the sub-matrix each PE is responsible for.

Parallel system	Latency	Bandwidth
Multi-core processor	107 clock cycles[2]	32 Bytes per cycle[3]
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 `Double`s, the latency will be the dominating contributor to the communication cost, not the bandwidth.

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 can only go in one direction in my assumed parallel system.

Distributed computer For this parallel system, I am assuming that we have some set of PEs that are distributed geographically across a continent, 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 a round-trip time latency of under 15 ms in the last 12 months, but use the generous upper-bound of 30 ms (Verizon, 2022). I also use the round-trip time such that communication can happen over protocols where acknowledgement messages are sent as well. For the bandwidth, I used the regional average broadband speed across Western Europe, which is 90.56 Mbps Howdle (2021).

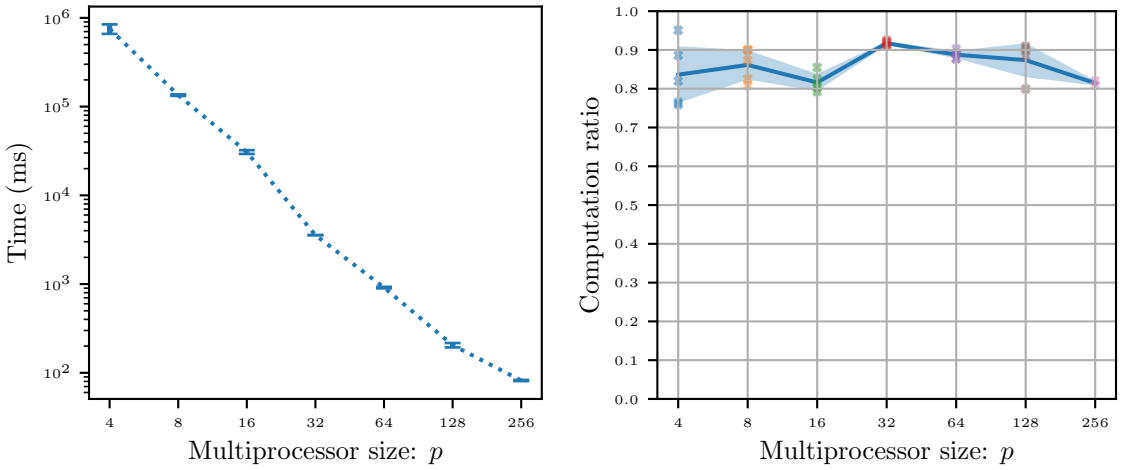
4.3.2 Advantage of parallel computation for solving APSP

Introduction. For the multi-core and supercomputer configuration, we see that the total execution time plot is flat or step-wise-like for small problem sizes, especially for large PE layouts. For example, the execution time is constant for $n < 100$ when we have 128×128 PEs. This is because we have enough PEs for each cell in the input matrix in this range, so increasing the problem size does not give more computation per PE. The step-wise-like increase in the execution time 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 and approach linear in the log-log-scale. This means that whenever we multiply the number of PEs by 4, we also decrease the computation time requires by a constant factor, although this factor seems to grow somewhat smaller the large the number of PEs.

[2]With the clock frequency of my system, this corresponds to 46.52 ns

[3]With the clock frequency of my system, this corresponds to 68.55 GB/s



(a) Total time

(b) Computation ratio

Figure 4.4: Compressed California road network $|V|= 1408$

Regardless, it tells us that for large problem sizes, there's always a benefit to adding more computation power as this reduces execution time.

We can also look at the parallel efficiency. As we saw in Equation 2.3, the parallel efficiency is the same as 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, which is in accordance with the last success criteria: "*The evaluation of the algorithm demonstrates that parallel computation gives a high parallel efficiency for solving APSP*".

In figure 4.3(c), we see that there is little benefit to parallel computation compared with serial until we reach large 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 latency constant, we eventually achieve a high parallel efficiency, but this only happens for the 4×4 and 8×8 arrangements for my selection of problem sizes, 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 increasing in a similar manner to that in figures 4.3(a) and 4.3(b). Therefore, we would expect the 128×128 configuration to overtake all the smaller ones eventually.

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

Conclusion

This chapter is empty!

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 Buluç and Gilbert (2010).

I also only considered one algorithm for solving APSP. It would be interesting to implement others on the simulation framework 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

Aydin Buluç and John R. Gilbert. Highly parallel sparse matrix-matrix multiplication. *CoRR*, abs/1006.2183, 2010. URL `http://arxiv.org/abs/1006.2183`.

Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David 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`.

Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, Peter Sanders, Lubo Brim, Jozef Gruska, and Jiri 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.

Jack 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`.

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. ISSN 0167-8191. doi: https://doi.org/10.1016/0167-8191(87)90060-3. URL `https://www.sciencedirect.com/science/article/pii/0167819187900603`.

Alex Fukunaga, Adi Botea, Yuu Jinnai, and Akihiro Kishimoto. A survey of parallel A. *CoRR*, abs/1708.05296, 2017. URL `http://arxiv.org/abs/1708.05296`.

Victor 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].

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

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

Jong Wook Kim, Hyoeun Choi, and Seung-Hee 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.

Vipin Kumar and Vineet 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.

Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua 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.

S.W. Moore. How to prepare a dissertation in latex, 1995.

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].

Piyush Sao, Hao Lu, Ramakrishnan Kannan, Vijay Thakkar, Richard Vuduc, and Thomas Potok. Scalable all-pairs shortest paths for huge graphs on multi-gpu clusters. 06 2020. URL https://www.osti.gov/biblio/1814306.

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

Appendix B

Proof of some theorem

B.1 Proof of that theorem

It's a tautology obviously.

B.2 Error bars

Behold, the plots with error bars:

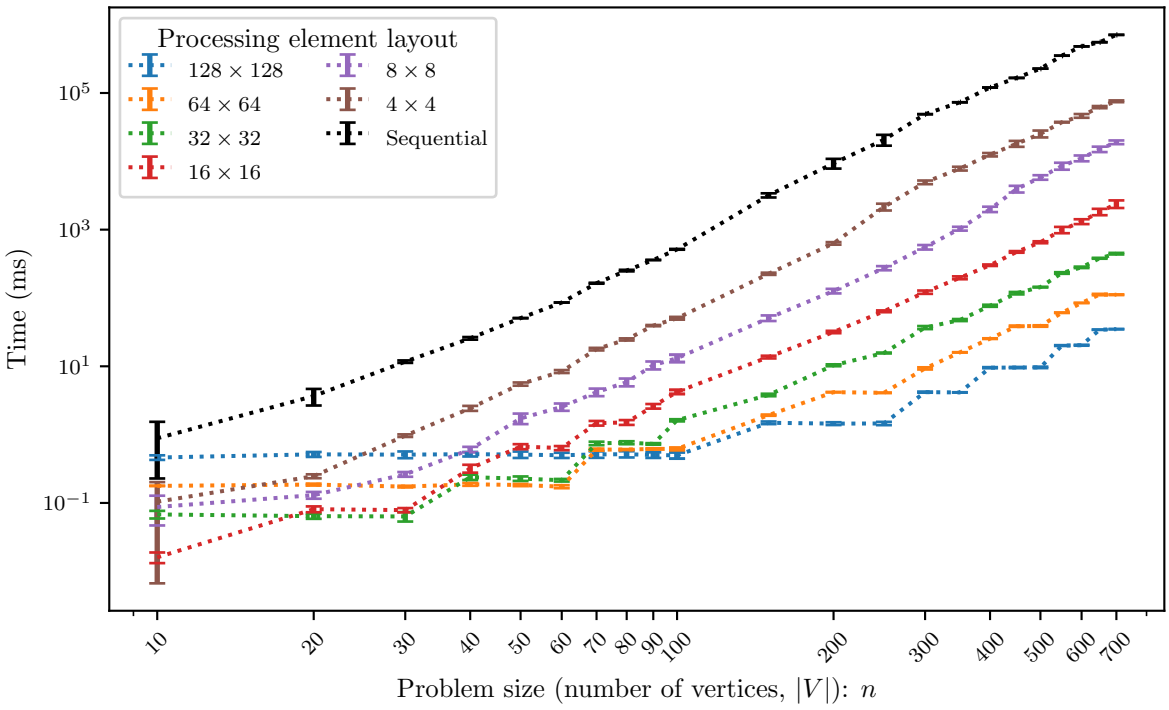


Figure B.1: Sandy bridge total time scaling

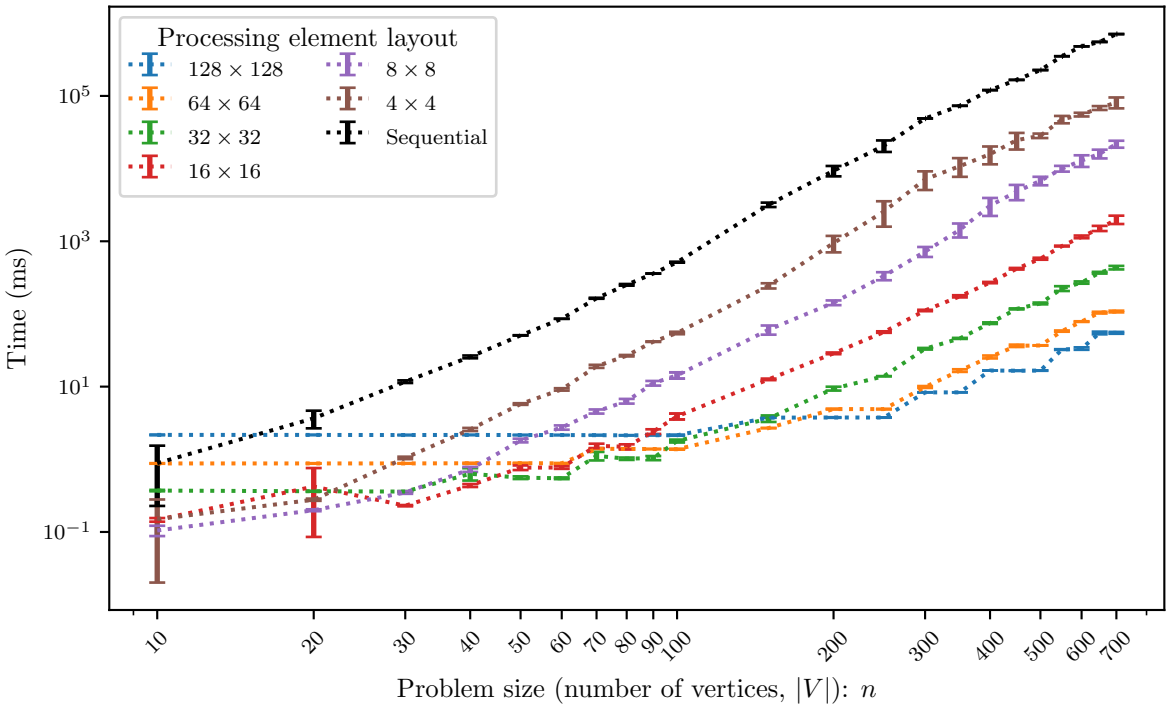


Figure B.2: Taihu-Light total time scaling

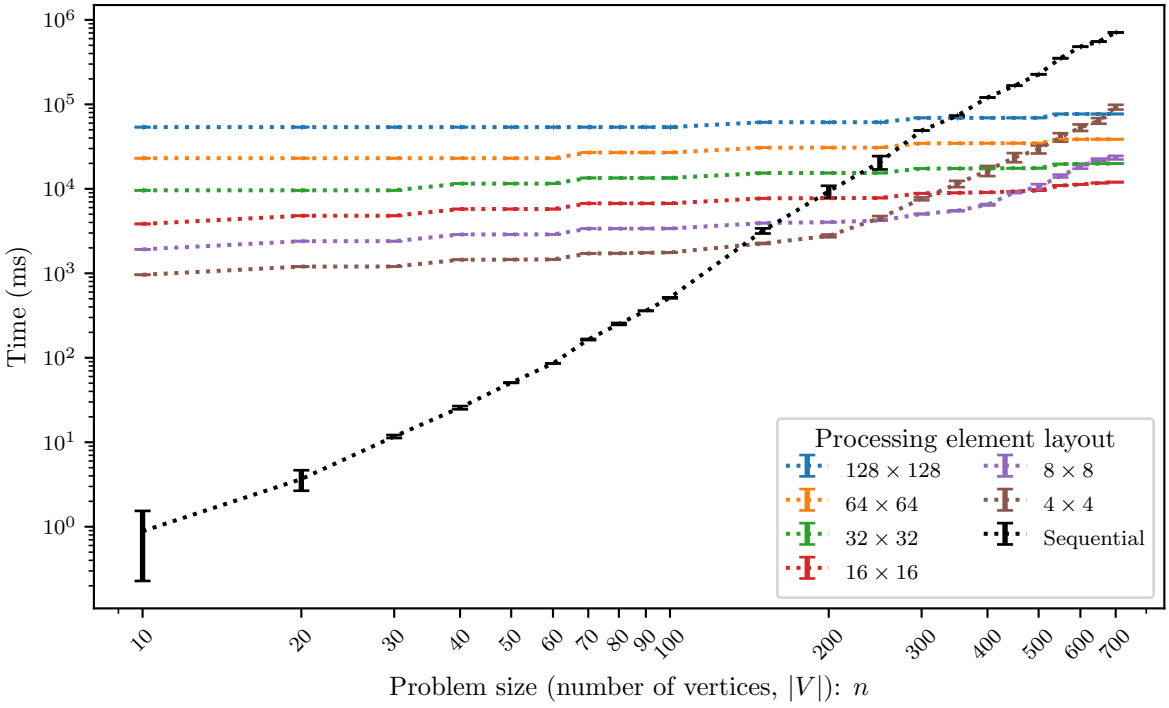


Figure B.3: Internet total time scaling

Appendix C

Project Proposal