Martin Richards

How to write a dissertation in LaTeX

Computer Science Tripos – Part II

St John's College

April 25, 2022

Proforma

Name:	**Martin Richards**
College:	**St John's College**
Project Title:	**How to write a dissertation in LaTeX**
Examination:	**Computer Science Tripos – Part II, May 2021**
Word Count:	**1587**[1] **(well less than the 12000 limit)**
Project Originator:	Dr M. Richards
Supervisor:	Dr Markus Kuhn

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.

Work Completed

All that has been completed appears in this 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.

Special Difficulties

Learning how to incorporate encapulated postscript into a LaTeX document on both Ubuntu Linux and OS X.

Declaration

I, [Name] of [College], being a candidate for Part II of the Computer Science Tripos [or the Diploma in Computer Science], 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 [signature]

Date [date]

Contents

List of Figures

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 Project structure

1.2 Motivation

1.2.1 Parallel algorithms

They are cool.

1.2.2 The all-pairs shortest path problem

The *All-Pairs Shortest Paths* (APSP) problem is famous, but why is it called APSP?

Chapter 2

Preparation

Introduction goes here.

2.1 Parallel computing

Parallel systems can be classified according to several different attributes. Here, I will ...

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 processing elements (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 Something something classes MPP and others

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

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

We consider a graph $G = (V, E, w)$, 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

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

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, giving the semiring $(\mathbb{R} \cup \{\infty\}, \cdot, \min)$. This operation is known as the distance product, and by repeatedly computing it for the weight matrix, we can find the length of shortest paths. If W is an $|V| \times |V|$ matrix, and $W_{ij} = w(e_{ij})$, the entries in W^k are distances between vertices in the graph G using 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). We can compute this matrix by squaring W $\lceil \log_2 n \rceil$ times. This is the idea behind our MatSquare algorithm. We also need to modify this algorithm to also keep track of the edges used in the shortest path, but more details of this will be explain in section 3.3.

2.2.3 Fox-Otto's technique

In Fox et al. (1987), they describe a technique for multiplying two matrices on a parallel distributed-memory system, such that the amount of communication is minimized. This is a suitable technique as we maximise our parallel efficiency by minimising the communication. If we are computing $C = A \times B$, we can visualise the memory movement ....

2.3 Parallel architecture assumptions

Having reviewed some of the different classes of parallel systems, as well as the general idea behind our APSP algorithm, we now review the assumptions I have made about the parallel system I will be developing an algorithm for. Regardless of system, it should have a configurable number of PEs, and this number can be very high as we might want to evaluate performance with many.

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

[2]See appendix section A.1.

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

Introduction.

Parallel arcitecture simulation framework? 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

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

revision control with `git` and GitHub, development, testing and evaluation on my own laptop with specs ...

Intellij IDE for Java,

github workflows, build my project every time I commit to `main` branch, to make sure code always in a valid state.

2.6 Starting point

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`

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.

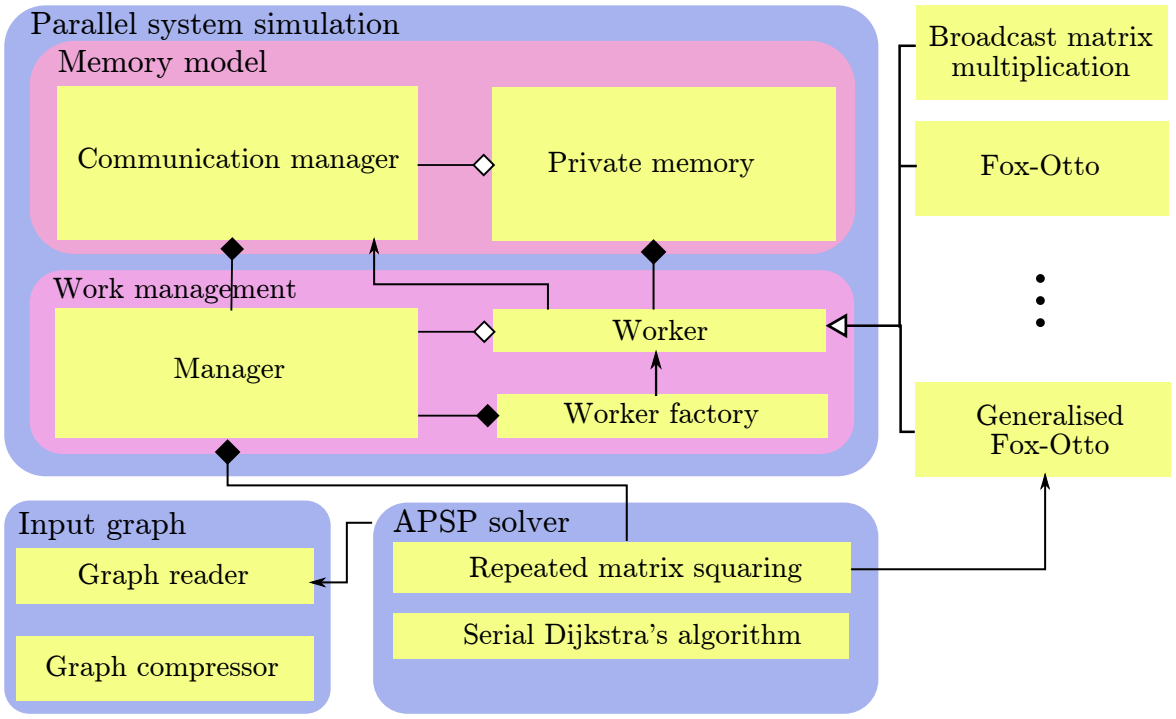


Figure 2.1: High-level overview of the implementation

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[4] standard. The various components were also unit-tested.

2.8 Conclusion

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

Chapter 3

Implementation

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 chose to use Erdös-Renyi graphs because they have previously been used in evaluation of APSP algorithms. I have tried to fit the parameters as closely to the real-world graph datasets as possible. 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 should have some correlation to the mentioned 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 graph is constructed by starting with a set of n vertices, and then independently include every possible edge e with probability p. This makes the edges follow the binomial distribution:

$$\deg(v) \sim \text{Binomial}(n-1, p) \tag{3.1}$$

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

Location	Average vertex degree
San-Francisco (SF)	2.549
North-America (NA)	2.038
? (OL)	2.305
California (cal)	2.061
? (TG)	2.614

Table 3.1: Average vertex degree for various road-network datasets[2].

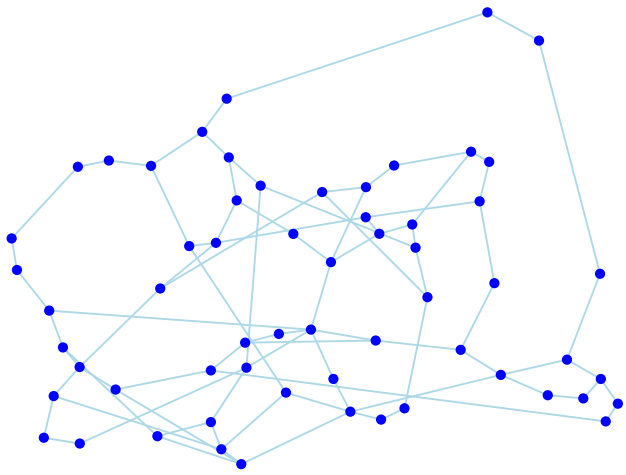


Figure 3.1: A visualisation of a graph generated with the modified Erdös-Renyi model. The graph has $n = 60$ vertices, and the length of the edges correspond to their weight[3].

with the average degree being $(n-1)p$. However, by instead starting with a *circular graph* and including each remaining edge independently with 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 3.1, I have plotted an example graph generated through this method. I then generated ? graphs of various sizes, from 10 nodes to 700 nodes for use in evaluation.

[2]These values were computed using the TODO??? method in `GraphReader`.

[3]The graph was visualised using the `ForceAtlas2` library in python. See TODO

3.2 Simulation of a distributed memory multiprocessor

3.2.1 Overview of components

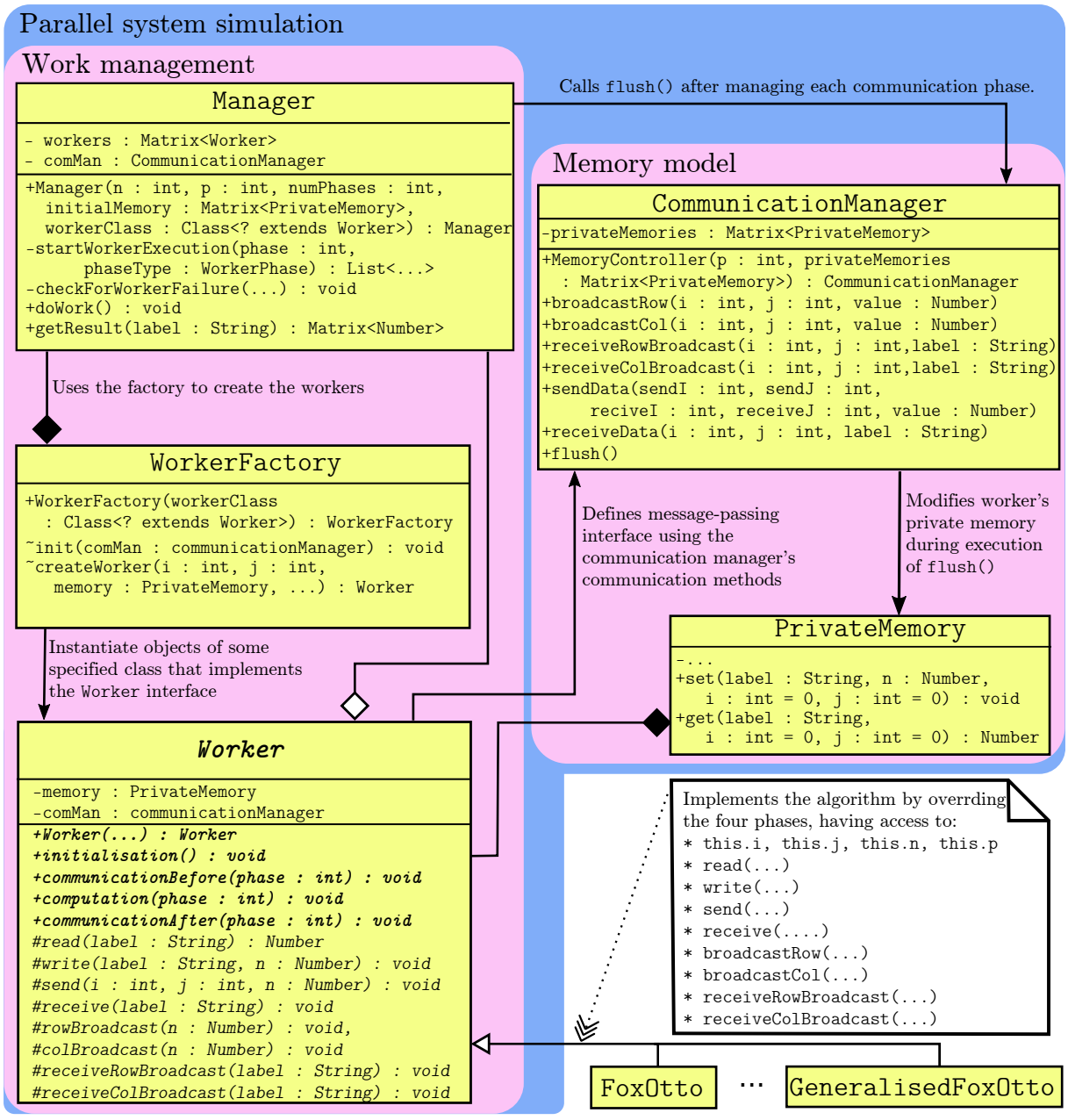


Figure 3.2: A overview of the classes and their interaction in simulator.

In Figure 3.2, we see 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 e.g. `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 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 3.3. 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 3.2 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 16^3 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

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

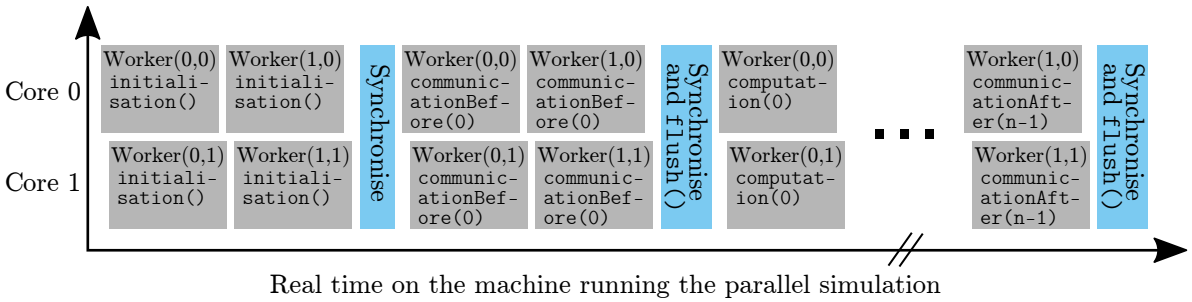


Figure 3.3: Example of multithreading when simulating a 4 processing elements on a host computer with 2 cores.

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 3.3. 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 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

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.

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 3.4, 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 $T_{(i,j)}^{(0)}$ 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

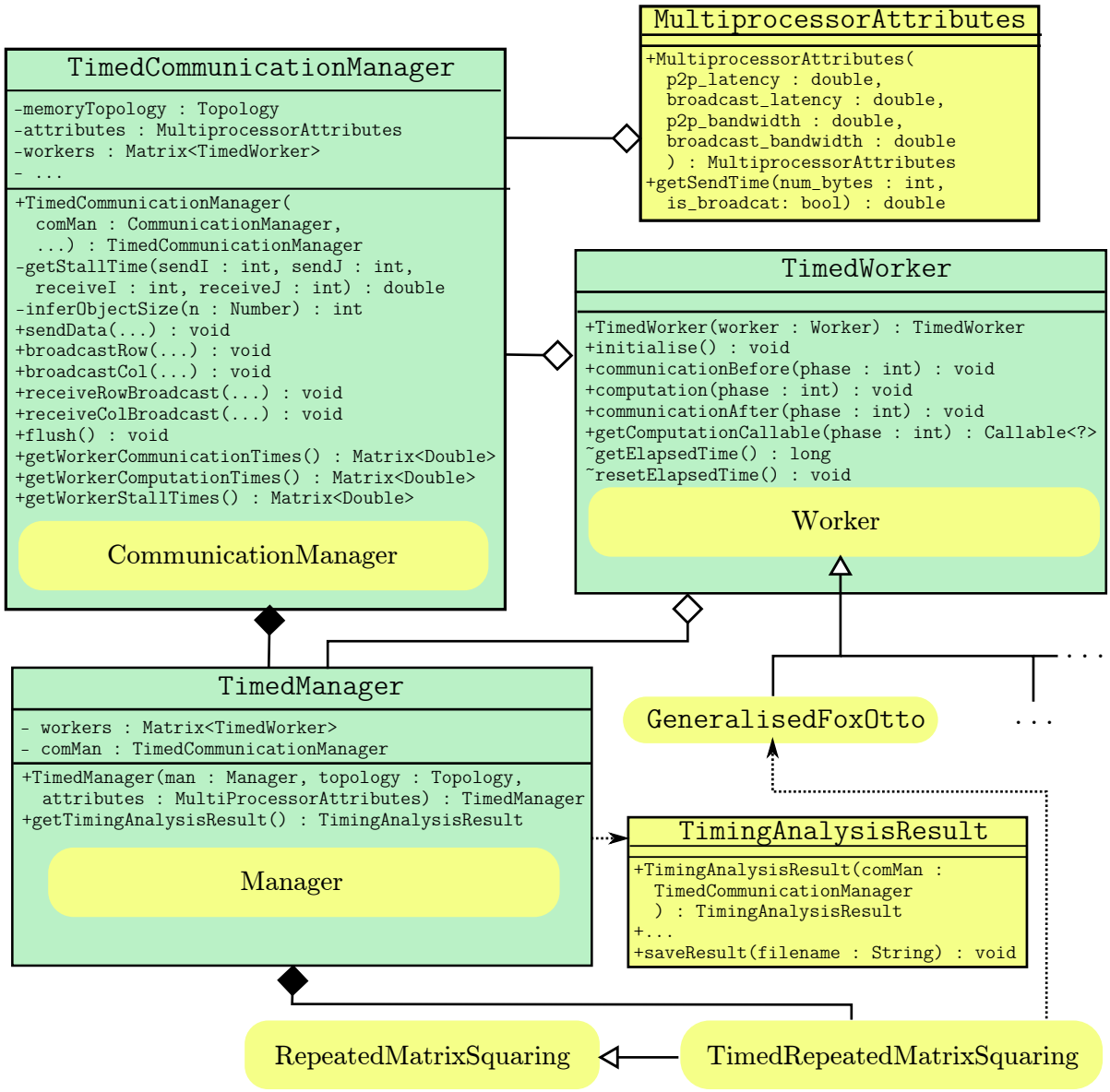


Figure 3.4: A overview of the components of the `timingAnalysis` package. Note that the `GeneralisedFoxOtto` and the `RepeatedMatrixSquaring` classes are not part of this package. Additionally, the green class diagrams with a yellow class inside it are shorthands for the decorator-pattern, where the outer class extends the yellow inner one and contains a reference to it.

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        */
computation(l);
t_0 ← current thread CPU time ;
for i = 0 ... num. repeats − 1 do
    computation(l);
end
t ← (current thread CPU time − t_0) / num. repeats
/* Disable read-only       */
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 3.4, 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

The *APSP via repeated matrix-squaring* algorithm, which I will abbreviate as *MatSquare*, is an algorithm. By using MatSquare, we can ...

To explain shortest paths are computed, we require two key definitions.

Definition 1. *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 2. *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[4].

Algorithm 3: MatSquare

```
Input  : Adjacency matrix W
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
```

The goal of the MatSquare algorithm is to compute the distance- and predecessor matrix. If we let W be the adjacency matrix for graph G, and set $W_{i,i} \leftarrow 0$ for each $0 \leq i < n$, we notice that it 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

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

However, we can reduce the number of distance product matrix multiplications from $\Theta(n)$ to $\Theta(\log n)$ by squaring W $\lceil \log n \rceil$ times, which is what I have done in algorithm 3.

To compute the predecessor matrix, we keep track of which intermediate vertex we use whenever we computer a distance product. When computing $M_{i,j} = \min_{0 \leq k < n}(M_{i,k} + M_{k,j})$ as part of the distance product $\otimes$, we refer to the used intermediate vertex as the witness for $w_{i,j}$.

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

Definition 3. *A witness matrix W for a distance product $M \otimes M'$ is a $n \times n$ matrix, where*

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

If find that $i \rightsquigarrow k \rightsquigarrow j$ is the next best path, we use the predecessor from the path $k \rightsquigarrow j$, as long as this path is not of length 0. This is what happens on lines 10 – 12 in algorithm 3.

The distance product, giving our running distance- and predecessor matrices, is computed in parallel using the parallel system simulation. We distribute the computation across p^2 PEs, where each PE executes the code in algorithm 4. The communication is based on a technique by Fox et al., initially used to just perform standard matrix multiplication (Fox et al., 1987). Given two $n \times n$ matrices A and B, we split them up into p^2 submatrices of size $\lceil n/p \rceil \times \lceil n/p \rceil$ and distribute them to the corresponding PEs. Before computation at each phase, we broadcast a submatrix of A along each row, as shown in Figure 3.5. After computation, we shift the B submatrices upwards, wrapping around the edges. This ensures that the corresponding entries to partially compute $C_{i,j}$ are available to PE (i, j) at each computation phase.

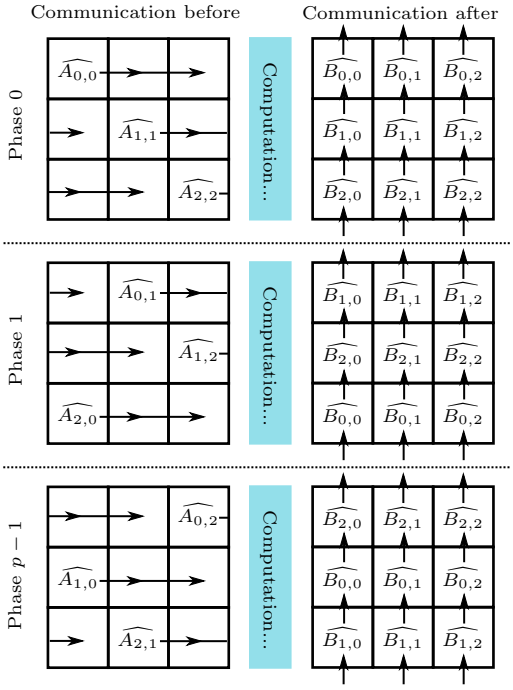


Figure 3.5: Message-passing pattern used in the Fox-Otto technique.

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 3.6. 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

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 **
```
$M'_{dist} \leftarrow n' \times n'$ 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**
    `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 $m = i_2, i_2 + 1, \ldots n', 0, 1, \ldots, i_2 - 1$ **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**
          $M'_{pred}[i_2, j_2] \leftarrow P[m, j_2]$;
        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

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. We have 5 cases, each of which gives different queries to the original graph:

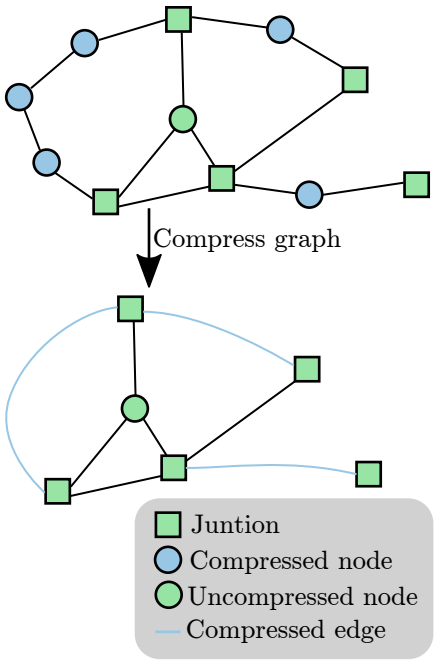


Figure 3.6: Graph before and after compression.

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:

$$p \overset{\text{nodes all in original graph}}{\rightsquigarrow} J_1 \overset{\text{nodes just in compressed graph}}{\rightsquigarrow} J_2 \overset{\text{nodes all in original graph}}{\rightsquigarrow} q, \tag{3.7}$$

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.

Chapter 4

Evaluation

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 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

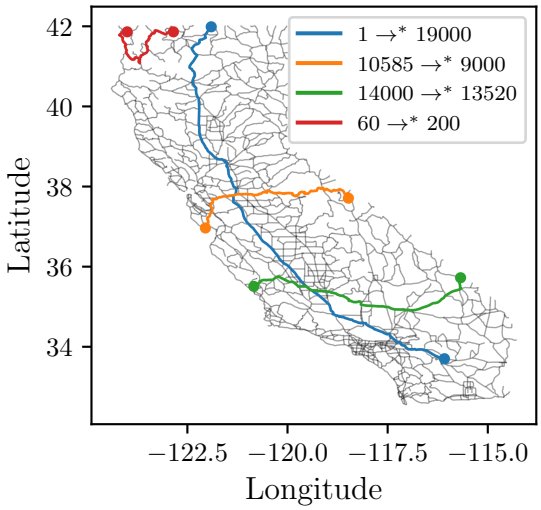


Figure 4.1: Four example shortest-paths in the Californian road network ($|V| = 21048$, $|E| = 21693$). These paths consist of 575, 328, 272, and 207 nodes, respectively.

[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.

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.

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. 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.

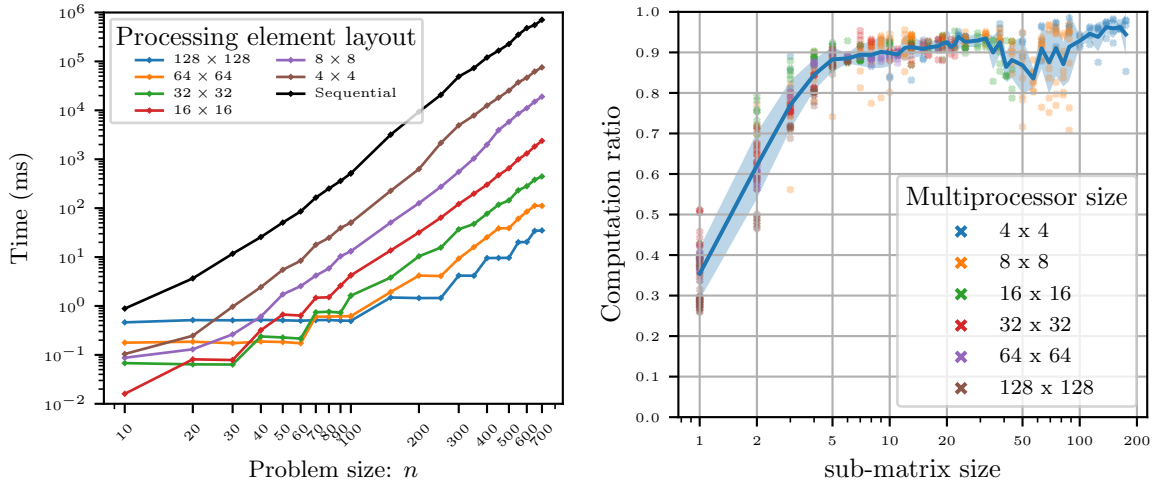
I also successfully "*[p]arallelised the matrix multiplication routine of the [MatSquare] algorithm to run on*" the parallel system simulation. This is done through my implementation of the `FoxOtto` and `GeneralisedFoxOtto` which use the `Worker` interface to create a parallel algorithm where coordination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

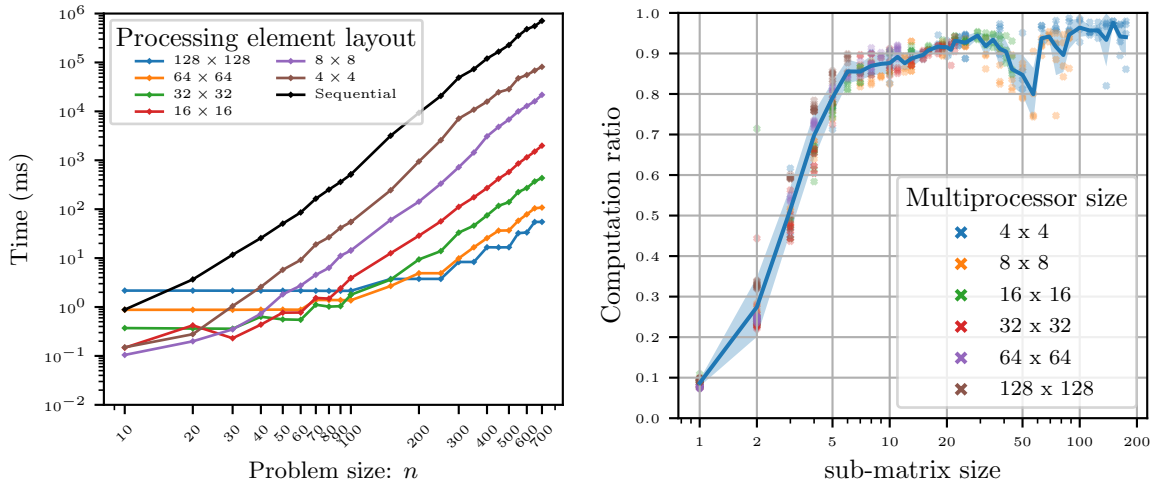
TODO

4.3.1 Choice of communication parameters

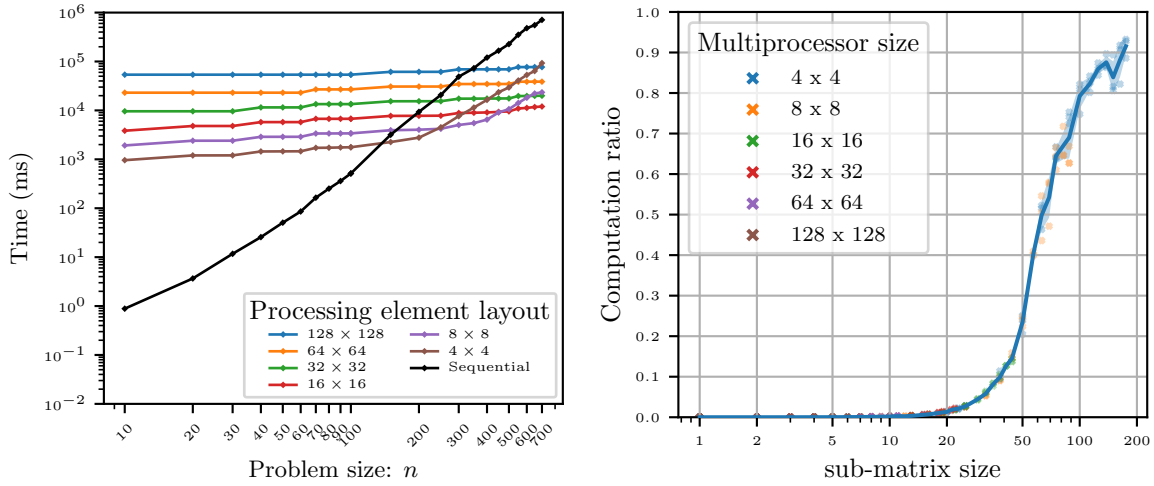
Multi-core processor The Sandy Bridge microarchitecture supports multiple cores which execute in MIMD fashion, and the L3 cache is shared across all cores while the lower-level caches are local to each core Sandy Bridge (client) - Microarchitectures - Intel (2020). For one core to send a data from its 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



(a) Multi-core processor (Sandy Bridge)



(b) Supercomputer (Sunway TaihuLight)



(c) Distributed computer (Internet)

Figure 4.2: 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.

sending data from one private L1 cache to another through the ring interconnect (using cache cohernecy 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 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

[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

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 4.2(a) and 4.2(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. 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 4.2 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 4.2(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 4.2(a) and 4.2(b). Therefore, we would expect the 128×128 configuration to overtake all the smaller ones eventually.

Chapter 5

Conclusion

This chapter is empty!

Bibliography

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.

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.

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.

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].

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:

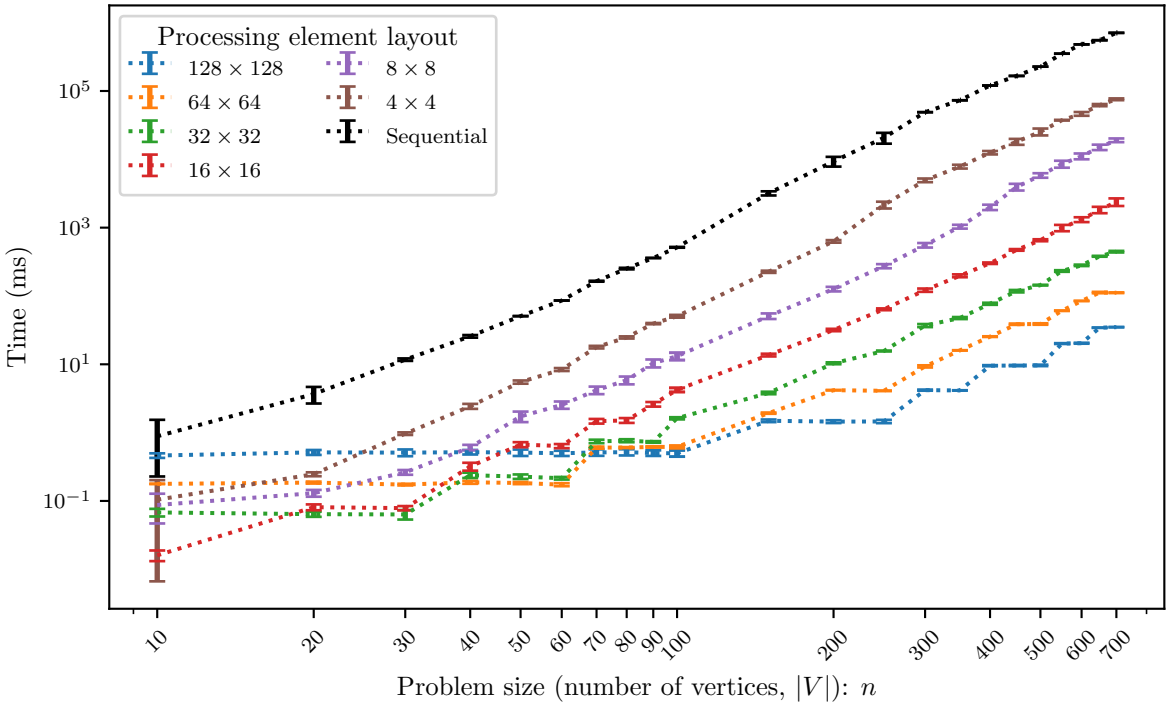


Figure B.1: Sandy bridge total time scaling

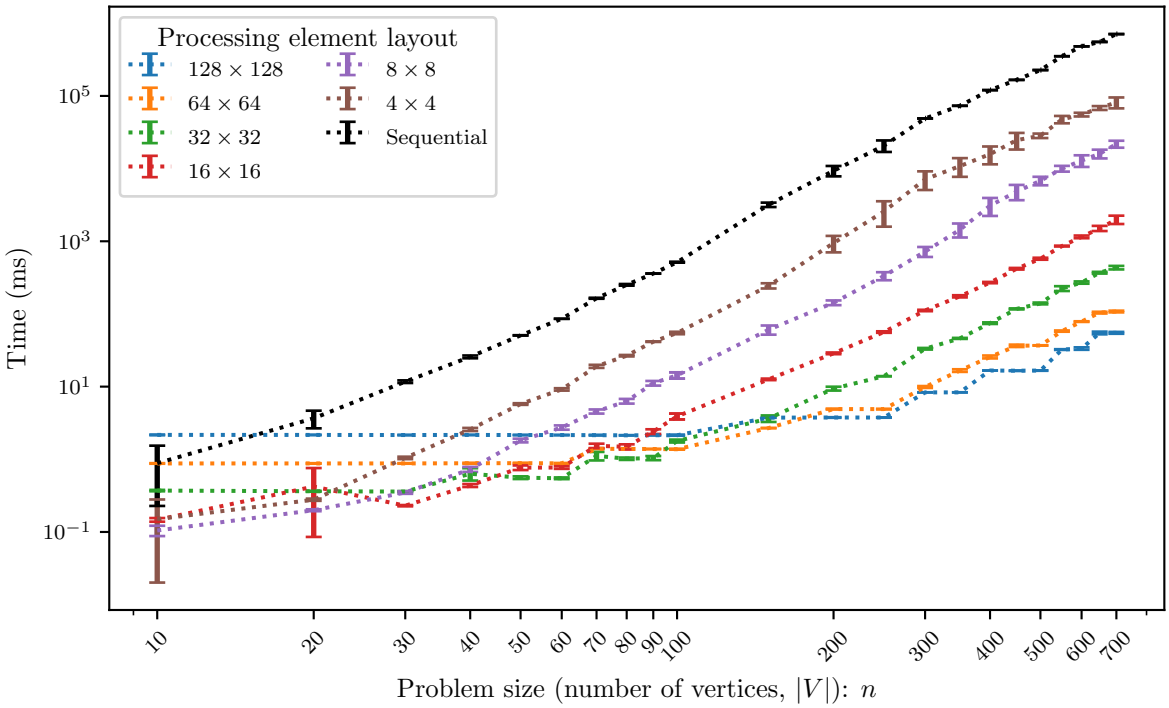


Figure B.2: Taihu-Light total time scaling

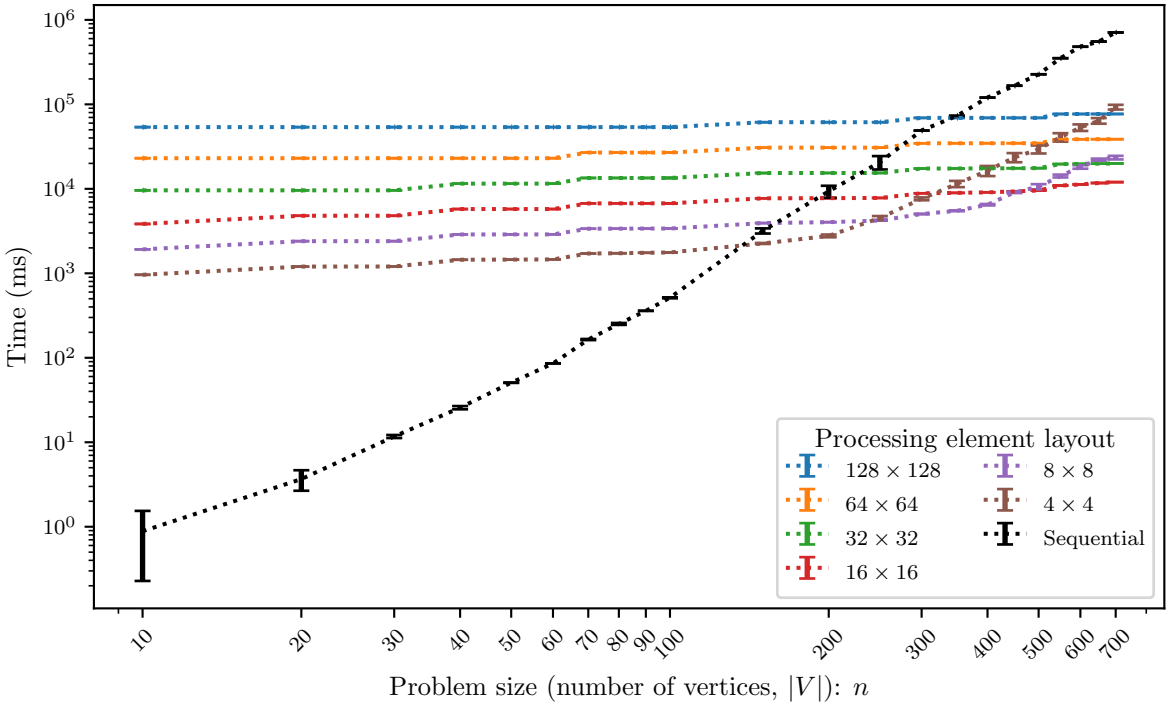


Figure B.3: Internet total time scaling

Appendix C

Project Proposal