Morton, John Magnus (2018) *JIT-based cost models for adaptive parallelism.* PhD thesis.

https://theses.gla.ac.uk/30753/

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses
https://theses.gla.ac.uk/
research-enlighten@glasgow.ac.uk

JIT-based Cost Models for Adaptive Parallelism

John Magnus Morton

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

August 2018

© John Magnus Morton

Abstract

Parallel programming is extremely challenging. Worse yet, parallel architectures evolve quickly, and parallel programs must often be refactored for each new architecture. It is highly desirable to provide performance portability, so programs developed on one architecture can deliver good performance on other architectures. This thesis is part of the AJITPar project that investigates a novel approach for achieving performance portability by the development of suitable cost models to inform scheduling decisions with dynamic information about computational and communication costs on the target architecture.

The main artifact of the AJITPar project is the Adaptive Skeleton Library (*ASL*) that provides a distributed-memory master-worker implementation of a set of *Algorithmic Skeletons* i.e. programming patterns that abstract away the low-level intricacies of parallelism. After JIT warm-up, *ASL* uses a computational cost model applied to JIT trace information from the *Pycket* compiler, a tracing JIT implementation of the Racket language, to transform the skeletons. The execution time of an *ASL* task is primarily determined by computation and communication costs.

The Pycket compiler is extended to enable runtime access to JIT traces, both the sequences of instructions and frequency of execution. Crucially for dynamic, adaption these are obtained with minimal overhead.

A low cost, dynamic *computation* cost model for estimating the runtime of JIT compiled Pycket programs, Γ, is developed and validated. This is believed to be the first such model. The design explores the challenges of estimating execution time from JIT trace instructions and presents three increasingly sophisticated cost models. The cost model predicts execution time based on the PyPy JIT instructions present in compiled JIT traces. The final abstract

cost model applies weightings for 5 different classes of trace instructions and also proposes a method for aggregating the cost models for single traces into a cost model for an entire program. Execution time is measured, and traces generated are recorded, from a suite of 41 benchmarks. Linear regression is used to determine the weightings for the abstract cost model from this data. The final cost model reveals that allocation operations count most for execution time, followed by guards and numeric operations.

The suitability of Γ for predicting the effect of *ASL* program transformations is investigated. The real utility of Γ is not in absolute predictions of execution times for different programs, but in predicting the effects of applying program transformations on parallel programs. A linear relationship between the actual computational cost for a task, and that predicted by Γ for five benchmarks on two architectures is demonstrated.

A series of increasingly accurate low cost, dynamic cost models for estimating the communication costs of *ASL* programs, K, are developed and validated. Predicting the optimum task size in *ASL* not only relies on computational cost predictions, but also predictions of the overhead of communicating tasks to worker nodes and results back to the master. The design and iterative development of a cost model which predicts the serialisation, deserialisation, and network send times of spawning a task in *ASL* is presented. Linear regression of communication timings are used to determine the appropriate weighting parameters for each. K is shown to be valid for predicting other, arbitrary data structures by demonstrating an additive property of the model. The model K is validated by showing a linear relationship between the combined predicted costs of the simple types in instances of aggregated data structures, and measured communication time. This validation is performed on five benchmarks on two platforms.

Finally, a low cost dynamic cost model, T, that predicts a good ASL task size by combining information from the computation and communication cost models ($\Gamma$and K) is developed and validated. The key insight in the design in this model is to balance the communications cost on the master node with the computational and communications cost on the worker nodes. The predictive power of T is tested model using six benchmarks, and it is shown to more accurately predict the optimal task size, reducing total program runtimes when compared with the default *ASL* prototype.

Acknowledgements

First and foremost, I would like to thank my supervisors, Phil Trinder and Patrick Maier. Your guidance, support and feedback have been crucial and greatly appreciated throughout this process.

I would also like to thank Blair, Stephen and everyone else who I've shared Office 421 with over the years.

Finally, I would like to thank my wife, Katie, for her love and unconditional support throughout.

Table of Contents

List of Figures

List of Tables

Glossary

AJITPar Adaptive Just-in-Time Parallelism. 14, 101

ASL *Adaptive Skeleton*. 14, 99

core A single processor, this term is usually used in the context of *multicore*, where multiple cores are on the same chip. 14, 21, 22, 24, 25, 28

irregular parallelism parallel programs where sub-tasks are interdependent or are of different relative sizes. 14, 18

JIT Just-in-time. 14, 18, 99, 101, *Glossary:* just-in-time

just-in-time a compilation technique which compiles code immediately before it is executed, in contrast to the traditional ahead-of-time compilation or interpretation. 14, 15, 18

PaRTE *Paraphrase Refactoring Tool for Erlang*. 14, 41, 42

performance portability the ability of a technique to bring performance gains on multiple platforms or architectures. 14, 17, 18, 99

static analysis analysis of a computer program that approximates properties of the program at execution time, but without executing it. 14

trace A record of the execution of a program. In terms of Just-in-time compilation, a trace starts at the top of a loop body and ends when execution leaves the loop.. 14, 18

transformation the rewriting of a part of or a whole of a program. 14

Chapter 1

Introduction

1.1 Context

The effects of *Moore's Law*[1] previously resulted in consistent increases in processor speed and a programmer would have to make little or no changes to their program to reap the benefits. Currently, *Moore's Law* manifests in an increase in the number of cores on a chip [2]. This, in principle, allows for continued increase in software performance on the new architectures, relying on software being rewritten to exploit the additional cores. *Multicore* programming requires writing programs which execute sections in parallel to each other. Multicore programming requires writing programs that execute code in parallel, and has proved both challenging and error prone. Many different approaches have been explored to address the issues. With a few notable exceptions, (e.g. OpenCL) most approaches target a specific hardware architecture. In consequence, code written in most parallel paradigms does not allow performance gains on one platform to be ported to another, or don't scale as well on other platforms. The problem of producing parallel software which *does* allow performance improvements to scale across architectures is known as *performance portability*. Once parallelism has been introduced, a further challenge is to select an appropriate task granularity. Too small a granularity and communication overhead can dramatically reduce performance, and too large often means that there are insufficient tasks to occupy all cores.

Just-in-time (JIT) compilation is a technology that allows interpreted languages to significantly increase their performance, often close to the speed of machine code. JIT compilation does not compile the entire program as it is executed, rather it compiles small parts of the

program which are executed frequently (these parts are described as "hot"). The most common compilation units are functions (or methods) and traces. *Trace-based* JIT compilation uses traces as a compilation unit. A trace consists of a series of instructions which make up the body of a loop. A complete trace contains no control-flow except at the points where execution leaves the trace.

The research was conducted as part of the EPSRC-funded Adaptive Just-in-Time Parallelism Project (AJITPar) [3]. AJITPar investigates whether performance portability for *irregular parallelism* can be achieved by using JIT technology and dynamically transforming the program for a particular architecture, using cost models of the traces executing on the architecture. The *Adaptive Skeleton* library (ASL) was created to test these ideas. *ASL* is a library of parallel algorithmic skeletons i.e. programming patterns that abstract away the low-level intricacies of parallelism. Programs written using these skeletons have their parallelism optimised using dynamic scheduling and adaptive transformation of the code at runtime. *ASL* applies information from the JIT cost models to the dynamic scheduling system after warm-up is completed.

The main value of the cost models in *ASL* would be determinism i.e. for a given program with a given input they will always produce the same cost value. This is in comparison to direct timings, which could contain noise. This is crucial for the way the default implementation of *ASL* applies the cost model (Section 2.7.1 — only three cost measurements are taken and compared against each other. If one of these measurements suffered from random variance, this would negatively affect any scheduling decision.

1.2 Thesis statement

This thesis asserts that it is possible to use the traces from a JIT compiler to predict the execution time of a program and, crucially, the effect on execution time of applying program transformations.

Additionally, the thesis will show how an effective, dynamic, type-indexed cost model for communications overhead can be constructed to accurately model communications overhead.

Finally, this thesis asserts that cost analysis of JIT traces combined with dynamic communications costing will allow *ASL* grouping and scheduling engines to automatically choose a good task granularity for some parallel architectures. This chosen granularity will result in significant speedup over the default *ASL* prototype.

1.3 Contributions

This thesis makes the following research contributions:

1. *The Development and Validation of the First Dynamic Computational Cost Model for JIT Traces,* Γ. Three increasingly parameteric cost models for predicting the execution time of JIT code are presented. Linear regression is used to parameterise the final version CM_w. The final parameter values show that memory allocation dominates the model on one hardware platform, but the reverse is true on another. Γ is shown to accurately predict the effect of applying program transformations. Extensions are made to Pycket to support runtime access to trace information, and to *ASL* to support the runtime application of Γ (Chapter 3) [4][5].

2. *The Development and Validation of the First Dynamic, Asymmetric, Bidirectional Type-indexed Communications Cost Model,* K. The development of the communications cost model, K, is described. The final version of K, K_{tbsd}, is shown to accurately predict the cost of serialisation, deserialisation and network transmission in *ASL*. The model K is shown to have an additivity property i.e. that it can predict the costs of arbitrary data structures from the costs of primitive types. K's ability to accurately predict arbitrary data structures is validated with five benchmarks on two architctures. K is demonstrated to meet *ASL*'s requirements for a useful communications cost model.

3. *The Development and validation of a Unified Cost Model of JIT Computation and Communication,* T. The execution time of a whole task is determined by its execution time and its communication overhead. The communication cost K, and the computation cost Γ are combined into a unified model T. T uses the insight that the communication cost on the *ASL* master node can be balanced with the communication and computation costs on the worker nodes. Experiments are performed with six

benchmarks that show T can be used to predict a good task granularity for *ASL* programs. This is shown to be between 17 to 54% better than the default *ASL* version. (Chapter 5).

1.4 Thesis Structure

The remainder of this thesis will be structured as follows.

- Chapter 2 provides the context of this work and details the architecture of *ASL*. A survey of related work on similar approaches to the problems of parallel programming and cost analysis follows.

- Chapter 3 describes the design and development of the first JIT-based computational cost model. It describes three increasingly parametric abstract cost models and details how linear regression techniques are used to parameterise them. The computational cost models are then evaluated on their ability to accurately predict the execution time effect of applying program transformations. The chapter also describes extensions made to the Pycket compiler to support access to the trace information.

- Chapter 4 illustrates the development of increasingly complex abstract cost models for modelling the communication cost of the AS system, followed by empirical deduction of a concrete version of our final model. Finally, we validate the model and validate that cost model instances for primitive data types can be combined to accurately predict cost models for compound data types.

- Chapter 5 presents a new model which combines the computation and communication cost models from detailed in Chapter 3 and Chapter 4. Finally, we show how this model can be used to optimise parallel throughput in the AS system.

- Chapter 6 summarises the work presented in this thesis, discusses limitations of the thesis and discusses potential avenues for future work.

1.5 Authorship

The work in this thesis has contributed to the following publications:

- J. M. Morton, P. Maier, and P. Trinder, "Jit-based cost analysis for dynamic program transformations, " *Electronic Notes in Theoretical Computer Science*, vol. 330, pp. 5–25, 2016.

  I was the lead author on, and main contributor to, this paper. I implemented the technology to extract the JIT traces, designed the computational cost model and validated its ability to cost transformations.

- P. Maier, J. M. Morton, and P. Trinder, "Jit costing adaptive skeletons for performance portability, " *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. ACM, 2016, pp. 23–30.

  I was a secondary contributor to this paper. The work in this paper relies on the JIT trace computation cost model, Γ.

1.6 Hardware Platforms

Throughout this thesis, two hardware platforms are used for experiments. The primary hardware platform is a Beowulf cluster named GPG, consisting of 16 2.0 GHz Xeon servers with 64 GB of RAM and gigabit Ethernet running Ubuntu 14.04. The secondary platform is named FATA, a 32-core 2.6Ghz Xeon with 64GB of RAM.

Chapter 2

Background

This chapter introduces key concepts in the fields of parallelism, programming languages and cost analysis. It also discusses related tools and details necessary technical background.

Section 2.1 discusses parallel hardware architectures in use today. Section 2.2 outlines various parallel programming approaches, covering a range of techniques from using POSIX threads, to distributed Haskell. Section 2.4 discusses program cost analysis, while Section 2.5 outlines program transformations. Finally, Section 2.7 outlines the AJITPar project and the architecture of *ASL*, and compares it to other relevant work.

2.1 Parallel Architectures

Computer architecture has developed rapidly since the beginning of the transistor era. The transistor density has increased exponentially, and previously resulted in a steady increase in sequential performance. Physical limits have been reached in the manufacturing and thermal performance of microprocessors and the increase in transistors is now manifested in increasing number of cores in single processors[2]. This is the reason that parallel programming is such a critical issue today. There are two general categories of parallel architectures discussed here, shared-memory — where parallelism takes place on a single local machine — and distributed memory — where parallelism could be distributed across a network. *ASL* is designed with both in mind. This section focuses on the *Multicore*, *NUMA* and *Cluster* parallel hardware, as these platforms are used in the work later in this Thesis.

2.1.1 Multicore

Most non-specialised processors sold today are *Multicore* processors, including many low-power and embedded CPUs [6]. The contemporary multicore CPU can be classified as a Multiple Instruction Multiple Data (MIMD) parallel processor[7]; this means that it enables multiple different instructions to be executed in parallel on possibly different data, while also allowing data parallelism of the form of single instructions operating on different data elements in parallel. The standard multicore processor has a number of levels of shared cache, as well as access to shared memory through a bus. Shared cache and shared memory allows significantly reduced communication overhead in a parallel program. Note that access to memory is *uniform*, in that access takes the same time regardless of which core access is requested from.

2.1.2 NUMA

As the number of cores in a processor increases, a single memory bus can become increasingly congested as tens or hundreds of cores are trying to access memory at the same time. One solution to this is *Non-Uniform Memory Access*(NUMA) [8][9]. In NUMA, memory access can be faster or slower depending on the locality of the memory - a core could have its own fast local cache and have access to separate shared memory. The advantage of this is that cores don't have to compete to access a single memory bus and can spend less time starved of resources. The main disadvantage with the NUMA approach is that it can be extremely costly to enforce cache coherence, the property that all caches in a multiprocessing machine have a consistent view of memory.

2.1.3 Clusters

A way of cheaply building or scaling a parallel system is to create a *cluster*[10][11], where multiple computers or processing entities are connected together, often using a network or other high-speed interconnects. The *Beowulf* cluster, first described by [11], is an example of a popular cluster architecture; Beowulf clusters consist of commodity hardware connected together in a local area network, usually using high-speed Ethernet. The primary measure-

ment platform in Chapters 3 to 5 is a Beowulf cluster of multicore machines. Parallel programs using a cluster would commonly be written using MPI [12].

The main benefits of a cluster system are cost, as a cluster can be built from cheap consumer systems instead of a massively expensive supercomputer, and scalability, since new nodes can be added as needed. The challenges with programming clusters are due to the lack of shared memory and overhead of sending data over a relatively slow network.

2.1.4 Other Parallel Architectures

The following hardware platforms form an important feature of the landscape of parallelism. However, they all require specialised tools or radical departures from mainstream general purpose programming, and are therefore considered out of the scope of this thesis. The use of a tracing JIT also precludes the use of many of these platforms, as a tracing JIT is not available or even possible. There are other exotic parallel platforms that are not discussed here, including microcore architectures such as the Adapteva Epiphany.

Graphical Processing Units

Graphical Processing Units (GPUs) are being increasingly used for parallel computation [13]. The high performance to price and performance to power consumption ratios drive the popularity of GPUs. The availability of programming languages and frameworks such as CUDA [14] and OpenCL [15] have simplified writing compute programs on GPUs. GPUs can have many cores and are particularly used for data parallelism and array programming. However, GPU programming is a completely different model of programming which involves explicitly transferring data between video RAM and main memory, is not natively supported by many mainstream programming languages, and parallelism offered by GPUs is almost exclusively data parallelism.

Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a popular architecture for signal and image processing. An FPGA can be programmed and reprogrammed by an end user using a Hardware Description Language (HDL) after it has been manufactured, unlike traditional inte-

grated circuits. This potentially allows the programmer to produce a circuit exactly appropriate to a particular problem and gives the *possibility* of massive parallelism. Unfortunately, the required use of a HDL to program the hardware, and the non-standard architectures produced as a result, renders FPGAs unsuitable as a target for a general purpose programming language. Compilers from high level languages such as C [16] and C++ [17] do exist, but the circuits produced by such compilers are inferior compared to that which a experienced hardware designer could produce in an HDL, using more circuit area and having lower clock frequency.

Manycore Processors

Tile processors are a relatively recent innovation where a single chip contains many processors as tiles in a two-dimensional array. Examples such as the Intel Single-chip Cloud Computer [18] and the Tilera TILE family [19] consist of many cores with 2D mesh networks interconnecting them. The main advantages are highly scalable performance and low power consumption. The Intel Xeon Phi [20] manycore processor is a relatively recent addition to the market.

Supercomputers

Supercomputers are massive parallel computers, consisting of often tens or hundreds of thousands of cores, connected by custom, high-performance interconnects [21]. The Sunway TaihuLight MPP [22] at the time of writing, leads the TOP500 list of supercomputers [23] with a performance of 93 TFlops, while a current commodity desktop multicore may have a performance of around 20 GFlops, indicating the massively improved throughput available on a Supercomputer.

These machines are often used for extremely intensive numerical computations such as Physics simulations like lattice Quantum Chromodynamics. Programs design for supercomputers are often extremely finely hand-tuned for specific architectures and thus are not suitable platforms for *ASL*

2.2 Parallel Programming

Parallel programming is often considered to be challenging. This is explained by the observation that parallel programming involves the management of parallel modification of shared state - problems arise when *tasks* — individual computations executed in parallel to other tasks — have an incoherent view of the state they are supposed to share. The difficulty in parallel programming is manifest in two areas: communication and synchronisation.

Even when communication and synchronisation problems are overcome, scheduling the tasks to make most efficient use of the resources is still an issue - parallelising tasks is of no benefit if it does not amortise the communication overhead, and if it does, we want to make the most efficient use of resources, especially given the time and energy cost of long running computations. This chapter will discuss programming language level approaches to solving these problems.

Emerging approaches offer ways of abstracting away and minimising the amount of state modifying code, in the case of pure functional programming; or by avoiding sharing state at all, in the case the actor model[24], where "processes" or "actors" pass messages to each other without sharing any explicit state. These "shared-nothing" approaches generally have the benefits of stronger scaling than shared-state, but at the cost of higher communication overhead. Note that these approaches are applicable to concurrency as well as parallelism.

As with other programming problems, there are a number of completely different approaches to managing this. Essentially, there are two fundamental models for parallel programming - *Task Parallelism* and *Data Parallelism* [9][8]. It is possible to view data parallelism as a subset of task parallelism, as it easy to model a data parallel system using task parallelism, perhaps using algorithmic skeletons[25].

2.2.1 Task Parallelism

One of the fundamental models of parallel programming is task-based parallelism[26], where, rather than the same instruction being applied to separate sections of data in parallel, multiple separate computations are executed in parallel.

Task parallelism is based around the concept of a *Task* - an independent computation execut-

```
void parDemo(){
    spawn a;
    spawn b;
    spawn c;
    d();
    sync();
    e();
}
```

Figure 2.1: Simple pseudo-C parallel program

ing in parallel to other tasks. These tasks could be executing on different cores on the same processor, or entirely different machines in a cluster. The main distinguishing feature of task parallelism is that these tasks can be completely separate computations or entirely separate functions or programs being executed.

Nearly all mature general purpose programming languages have some support for task parallelism e.g. Java has a fork/join framework [27], C++ has OpenMP [28] and MPI [12] amongst others and Haskell has GpH [29].

For a motivating example, consider the code in Figure 2.1. This code is written in hypothetical C-like language with basic task parallelism. The keyword `spawn` spawns a given function in parallel, while the call to `sync` blocks progress until the parallel tasks are complete. This explicit synchronisation is typical of task parallelism.

ASL and the work in this thesis are concerned with task parallelism.

Algorithmic Skeletons

One of the biggest problems when creating task parallel software is adapting an existing sequential algorithms to parallelism. The *Algorithmic Skeletons* approach developed by [25] aims to separate the algorithm from the code needed to parallelise it. These *skeletons* can be seen as design patterns for parallelism, including such common parallel paradigms as, map, fork/join, divide and conquer and pipelining. It is also possible to represent both data and task parallelism with algorithmic skeletons. It is possible to build algorithmic skeletons using evaluation strategies. Well-known examples of the map skeleton - where a data set is split into chunks and parallel tasks executed on each subset - include Google's MapReduce [30] and Apache's Hadoop[31]. Intel's Threading Building Blocks [32] is a C++ template li-

brary which provides algorithmic skeletons. Parts of the work in this thesis use Algorithmic Skeletons e.g Chapter 3.5.

2.2.2 Data Parallelism

In contrast to task parallelism, data parallelism [26] avoids distributing independent computations across Processing Elements (PEs), and instead distributes chunks of data and each PE performs the same computation on its own chunk of data. The Single Instruction Multiple Data (SIMD) hardware architecture [9] is an example of this, where a single instruction is executed in parallel over multiple items of data. Often, data parallelism is implemented in the form of arrays, matrices or vectors with special operators which perform operations over the data structure in parallel, such as matrix multiplication or the map operation.

The main drawback of data parallelism is that it is not a convenient model for some parallel problems. It is difficult to express a program in terms of data parallelism when the parallelism available in the algorithm in question is tied to its control flow rather than its data.

2.3 Parallel Languages

Nearly every programming language and mainstream operating system has some support for parallelism, and there are a plethora of software libraries and tools that support it. This section will discuss some of the most important and well known parallel languages.

2.3.1 Criteria

There are a number of criteria that can be used to classify parallel programming tools and languages, the most important are the explicitness of the language and whether it supports shared or distributed memory. Table 2.1 shows an overview of how sample parallel programming languages fit in to these criteria.

Language	Implicit/Explicit	Shared/Distributed Memory	Functional/Imperative
POSIX Threads	Fully Explicit	Shared	Imperative
OpenMP	Semi-Explicit	Shared	Imperative
MPI	Fully Explicit	Distributed	Imperative
PGAS	Semi-Explicit	Shared	Imperative
GPH	Semi-Explicit	Shared	Functional
HDPH	Semi-Explicit	Distributed	Functional
SAC	Implicit	Shared	Functional
ASL	Semi-Explicit	Distributed	Functional

Table 2.1: Properties of some Parallel Programming Languages

Explicit/Implicit

Some languages and tools require the programmer to manually specify every aspect of parallelism such as scheduling, communication and synchronisation, while at the other extreme, some languages and tools automatically parallelise programs. The degree to which manual programmer intervention is required is known as *explicitness* and is described by [33] and [34].

A *fully explicit* model requires the programmer to manually specify the scheduling of tasks on PEs and the synchronisation and communication of these tasks too, while *semi explicit* models have one or more of these aspects automated while the programmer handles the others. *Implicit* models require minimal intervention from the programmer, and the compiler or run-time attempts to parallelise and synchronise the code. *Fully implicit* models automate all aspects of parallelism, and can be capable of parallelising an unmodified sequential program without any programmer intervention.

Shared/Distributed Memory Paradigms

Many parallel programming paradigms assume access to shared memory. The main benefits of a shared memory approach are a simple programming model and low communication and synchronisation overhead, while scalability is often limited - synchronisation costs will increase with the number of cores.

In contrast to shared-memory approaches, distributed-memory parallel programming models assume that individual processing elements do not have shared access to memory. This poses a new set of problems to the programmer attempting to build a parallel system compared

to a shared memory system, as the programmer has to be concerned with vastly increased communications costs.

Imperative/Functional Programming

Functional programming languages - e.g Haskell, ML, Lisp — differ from imperative languages in that they restrict mutable state, treat functions as first class values, and often (but not always) have rich type systems e.g Haskell, ML. The main advantages of parallel functional languages are the same as those in sequential functional languages; that the type system or restricted mutability allows the programmer to better able to reason about the program and thus it can be easier to safely parallelise.

2.3.2 POSIX Threads

POSIX threads (known as `pthread`) [35] are a standard interface for concurrent programming, but can be used as an explicit, shared memory parallel programming model and is available on nearly every mainstream operating system. It has also influenced the thread programming model in many other languages, such as Java [36].

The fundamental unit of the `pthread` susbsystem is the *thread*. Threads can be described as a computation executing in parallel or concurrently, and can be considered an implementation of the task abstraction discussed in Section 2.2.1. They have their own stack, but are part of the same operating system process that spawned it. True POSIX threads are not a language or external library feature, but are part of the operating system and the OS has responsibility for scheduling them.

The `pthread` library also exposes mutual exclusion lock structures and system calls for managing them to enable the synchronisation of threads and safe access of shared resources. Unfortunately, these are easy to misuse and careless programming can result in deadlock or livelock [37].

Threads are used when low-level parallelism is required and are almost a necessity when performing I/O in C or other low-level languages. However, they may be considered too low-level for most highly parallel systems and open the door to major, difficult to debug prob-

lems. Other tools, such as OpenMP, offer higher-level, less fragile constructs and scheduling options.

POSIX threads can be classified as a fully explicit approach, as they require the programmer to specify communication between threads and the creation and scheduling of them.

2.3.3 OpenMP

OpenMP [28] is a semi-explicit, shared-memory library and language extension for task parallelism. OpenMP is one of the most widely used shared-memory parallel models [38]. OpenMP provides a number of compiler pragmas which cause marked code to be executed in parallel. It can be described as a fork/join model. A number of primitives are provided for synchronisation of threads. Directives exist for specifying atomicity of memory access, mutual exclusion and barrier synchronisation. OpenMP also allows for restrictions and control over the sharing of data between threads, such as the `private` pragma which causes data to be private to each thread. OpenMP does have some support for data parallelism, in the form of parallel `for` loops. OpenMP is specified for C, C++ and FORTRAN, but bindings or implementations exist or are planned for other languages e.g. Java [39].

2.3.4 MPI

MPI or Message Passing Interface [12] is a fully-explicit, distributed memory parallel programming interface. The MPI interface allows programs running on different nodes to send messages to each other in point to point and broadcast modes. MPI requires that types are specified in messages and allows the programmer to define new cross-platform types. MPI is very widely used in scientific computing, where it is the de-facto parallel programming model [40].

2.3.5 PGAS Languages

A Partitioned Global Address Space (PGAS) is a semi-implicit shared memory parallel programming model (shared memory is also supported, but is not the main focus). In a PGAS

language, such as Co-Array Fortran [41] or Unified Parallel C [42], the address space is partitioned so that each thread has their own local partition, which they act on in parallel. An advantage of this approach is that each local address space will be more local to the CPU, resulting in improved performance.

2.3.6 Programming Language Specific

The following parallel languages are different from the previously described as they are specific to particular programming language implementations, or, in the case of Single Assignment C, are programming languages themselves.

GPH

By default, Haskell has no support for parallelism other than `IOThread`, which adds lightweight thread based concurrency.

Glasgow Parallel Haskell (GPH) is a functional, semi-explicit, shared-memory parallel programming model. It is implemented as a Haskell extension (which is now part of GHC) that adds side-effect free task parallelism to Haskell through the `par` combinator, which spawns evaluation of an expression in parallel in a purely functional manner. Unfortunately, `par` alone is not enough, since lazy evaluation can stop expressions from evaluating in parallel if they are immediately used. To fix this, `seq` is introduced, which forces the evaluation of an expression in a particular order, giving the expression spawned by `par` a chance to evaluate in parallel.

Building complex parallel expressions using `par` and `seq` can be tricky and if done wrong can easily result in a sequentially evaluated expression. An alternative is to use the `Par` monad[43], a monadic approach to parallelism which allows the evaluation to be expressed in an imperative style.

HdpH

There are many different parallel Haskell implementations[44]; so far, this chapter has only discussed shared memory approaches.

HdpH[45] is a domain specific language for expressing distributed task parallelism in Haskell. It uses a `Par` monad to encapsulate its operations and has a data type for representing serialisable closures. Structures called `IVars` are used for communications between PEs and functions exist for explicitly pushing computations to a PE or implicitly sparking a computation in parallel. Scheduling and load management is dealt with implicitly, with idle nodes using work stealing.

SAC

Single Assignment C[46] (SAC) is a pure functional language based on C. It has support for, and is based around, high-performance array operations. It has data-parallel operations on arrays and matrices which are shape-polymorphic, meaning that they work independent of the dimensionality and regularity of the arrays. It can be described as an implicitly parallel language.

Cilk

Cilk [47], a shared memory parallel programming language, uses work stealing scheduling. The scheduler uses an optimisation heuristic known as "work first", where tasks are spawned repeatedly before being executed. Thus, the scheduler attempts to minimise overheads of computation rather than other overheads.

2.4 Resource Analysis

Resource analysis, or cost modelling, is important in resource-limited systems like most embedded systems, in hard real-time systems where timing guarantees are required, and for directing program refactoring or transformation. This thesis seeks a dynamic resource analysis to inform dynamic program transformations. Recently there has been significant progress in both the theory and practice of resource analysis. Some of this progress is driven by the TACLe EU COST action [48] and reported in the FOPARA workshops [49]. In the context of parallel programming, resource analysis can be used to determine whether it is

worth introducing parallelism e.g the cost analysis of a piece of code shows that its execution time will not be long enough to amortise the cost of parallelising the code.

Cost analyses can build on static analysis techniques, important examples of which include *dataflow analysis* [50], which analyses how values are assigned to variables and are propagated through the program; *control-flow analysis* which analyses the order, looping and branching of program statements/expressions; and abstract interpretation, which interprets an abstraction of the programming language e.g. an abstract interpretation where all semantics were abstracted away and the interpretation is only concerned with memory access patterns.

Analysis techniques exist for a range of program resources, for example execution time [51, 52, 53], space usage [54, 55], or energy [56]. The resource of interest here is predicted execution time. For many applications, e.g. embedded and real-time software systems, the most important performance metric is *worst case execution time*. Various tools [51, 57, 52] have been built to statically estimate or measure this; an example is aiT [57] which uses a combination of control flow analysis and lower level tools, such as cache and pipelining analysis. Cache and pipelining analysis attempts to predict the caching and processor pipelining behaviour of a program and is performed in aiT using abstract interpretation. Here, however, expected, rather than worst case execution time is predicted. Moreover, precise absolute costs are not needed: approximate relative costs should suffice to allow the transformation engine to select between alternative code variants.

A range of analysis techniques are used to estimate the resources used by programs. High level cost analysis can be performed on the syntactic structure of the source code of a program, e.g. using a mathematical function of C syntactic constructs to estimate execution time [58]. Low-level representations of code and bytecode can be used as source for static resource analysis [59, 60, 55, 61, 62]. For example the COSTA tool [60] for Java bytecode which allows the analysis of various resources using parameterized cost models, and the CHAMELEON tool [61] which builds on this approach and uses it to adapt programs.

There are many other approaches in cost analysis including amortized resource analysis [63, 62], incremental resource analysis [64], and attempting to enforce resource guarantees using *proof-carrying code* [62, 65] (the MOBIUS project is a prime example).

Control flow is a key element of many resource analyses [51, 57]. However, as JIT traces do

not contain any control flow, these types of analysis are redundant and a far simpler approach will suffice. This is fortunate as the static analysis must run fast as part of the warm-up phase of the execution of the JIT compiled program.

2.4.1 Parallel Resource Analysis

There is a large body of work that applies resource analysis to parallelism [66, 59, 53], which has produced several examples of parallel cost models. Perhaps the simplest parallel cost model is the Parallel Random Access Machine (PRAM) [67] model of parallel computation. This model assumes a parallel machine where processors can perform parallel operations on shared memory, with synchronised access. Cost analyses using this model allow for basic estimates of parallel speedup on a multicore shared memory system, but is less useful for NUMA systems or distributed memory systems.

The LogP cost model [68] attempts to address these shortcomings by taking into account the latency of communication between processors, the overhead of sending messages and the minimum gap between sending them, as well as the number of processors.

Other parallel cost models include the Bulk Synchronous Processes model [69], and several extensions to the Bird-Meerten's formalism [70], including the work by Rangaswami [71].

2.5 Program Transformation

Static analysis and cost modelling can provide much useful information about a program, but this information is only useful if acted upon, and static analysis is often performed with automatic code transformation in mind. In general, the term program transformation covers everything from translation of a source program in one language to another to low-level optimisations on machine code like peephole optimisations. In AJITPar, code transformations the parameters of skeletonized code are modified based on information gathered by the cost analysis.

[72] describes the different types of translations as high-level source-to-source translations, *synthesis* - where programs are transformed into a lower level of abstraction (compilation is an example), reverse engineering, *normalisation* - where a program is transformed into a

syntactically simpler program in the same language, *optimisation* where a program is transformed to increase some aspect of the performance of the program, *refactoring* and *renovation* - where a program is brought up to date or fixed in some way.

Essentially, code transformation systems can be seen as a Term Rewriting System, just with different rewrite rules and strategies. Transformation strategies described by [72] include sequential composition of rules, speculative application, traversal over abstract syntax trees and application strategies where context is carried and acted upon. Strategies are necessary since applying the rewrite rules to an entire program until every part is in normal form (with respect to the rewrite rules) may not terminate or be confluent.

Program transformations are central to optimising compilers. GHC, for instance, aggressively optimises Haskell code by equational rewriting [73, 74]. Transformations can also be used for optimising for parallel performance. Algorithmic skeletons [25] – high level parallel abstractions or design patterns – can be tuned by code transformations to best exploit the structure of input data or to optimise for a particular hardware architecture. Examples of this include the PMLS compiler [75], which tunes parallel ML code by transforming skeletons based on offline profiling data, and the Paraphrase Project's refactorings [76] and their PARTE tool for refactoring parallel Erlang programs [77]. PMLS is an automatically parallelising compiler for Standard ML which turns nested sequential higher-order-function calls into parallel skeleton calls and performs code transformation based on runtime behaviour of subparts of the program.

2.6 Just In Time Compilation

Most programming language implementations can be classified into either 'compiled', where code is compiled into machine language, or 'interpreted', where code is executed line-by-line or is compiled to a lower-level representation, such as a bytecode representation or even just the abstract syntax tree, which is then executed by a virtual machine.

The key benefit of the virtual machine interpreter model was that a degree of portability could be achieved without having to write a full static compiler for each platform. However, they are generally slower than statically compiled machine code, due to the overhead of decoding the bytecode or other representation and the overhead of using a virtual machine to execute

the instructions.

A way of increasing performance is to dynamically compile the bytecode to machine language at runtime, and then execute the machine code instead of the bytecode. Ideally, the performance boost from executing machine code should offset the cost of dynamic compilation. This is known as Just-in-Time or JIT[78] compilation. JIT compilers retain the portable execution of interpreted code, while attempting to achieve the performance of native machine code. This could be considered difficult due to the time-dependent nature of JIT compilation: the same optimisations performed by static compilers are not feasible, and the reduced scope of the compilation reduces the information available to an optimiser. However, the JIT compiler can make use of dynamic information to perform optimisations that a static compiler cannot. *ASL* and the work in this thesis use the tracing JIT-compiled Pycket compiler.

2.6.1 Compilation Units

Simply compiling the entire program at runtime is obviously impractical, since it is essentially the same as statically compiling the program. At the other extreme, compiling at the level of a basic block is unlikely to result in much benefit due to the extremely limited scope for optimisation, limited benefit for the compilation overhead and the overhead involved in flitting between bytecode execution and native execution constantly. It is apparent that anyone implementing a JIT compiler needs to decide on an appropriate compilation unit for the JIT.

A popular choice is method or function JIT, where entire methods or functions are compiled. This allows easy interaction between native code and bytecode, as well as offering an acceptably large compilation unit.

Another option, though not as popular, is to use a trace. A trace is a sequence of instructions with one entry point but multiple exit points and this sequence can span many functions or methods. Traces can be selected which are always of an acceptably large size and which are always loops. Traces have no control flow in them (with the exception of a jump back to the beginning of a loop) so optimisation and compilation is much simpler than functions or methods which can contain arbitrary control flow. Lambdachine [79], LuaJIT [80] and PyPy [81] are examples of trace-based JIT compilers.

Even with a large enough compilation unit, a compiled trace or method may not be executed enough times to amortise the compilation cost. However, if the VM knows that the method or trace is repeatedly called then it is more likely to be worth compiling. A method or trace which is well-used is described as *hot*. [82] describe the Java HotSpot compiler which detects "hot spots" in the bytecode and schedules "hot" methods for JIT compilation.

2.6.2 Existing JIT Compilers

Java's hotspot compiler[82] is perhaps one of the most well known JIT compilers. It uses methods as its compilation unit.

Lambdachine [79] is an experimental trace-based Haskell JIT compiler based on LuaJIT, which is notable as Haskell is known as a language normally statically compiled to native code. [79] reported that performance could occasionally match the performance of the Glasgow Haskell Compiler.

PyPy[81], is a trace-based JIT implementation for the Python programming language, as well as a compiler generating tool-chain - a compiler implementer can write an interpreter using PyPy's RPython language and the tool-chain generates a full interpreter (optionally with a JIT). PyPy's JIT is notable for the fact that it traces the main loop of the interpreter itself, rather than the user program, a technique described by [81] as *meta-tracing*. The front-end compiler implementer provides annotations to the JIT which specify the start points of user program loops. When the JIT meets one of these points, recording of a trace begins; when the interpreter reaches the same annotated point again, it finishes the recording and compiles the trace when it gets hot enough. Performance of PyPy's python interpreter can often meet or exceed the performance of the reference Python interpreter.

Pycket[83], is an implementation of the Racket language (a form of Scheme) on PyPy's tool chain, using the PyPy meta-tracing JIT. Results showed that the implementation's performance was comparable to Racket and other scheme implementations. *ASL* is built on a version of Pycket modified to support TCP/IP communication and runtime costing of JIT traces.

2.6.3 Parallelising JIT

A trace of a program can yield useful information about possible optimisations that can not be discovered statically. Similarly, a trace can be used to glean information about dependencies between variables which may not be possible to determine statically; analysis is also simpler due to the lack of control flow within a trace. Furthermore, if a trace is hot, it shows that that trace would likely benefit from being parallelised. Since a trace often forms the body of a loop, it is possible to parallelise a trace using data parallelism techniques [84].

Sun *et al*. describe a parallelising compiler using traces [85]; the system described is built on top of an actual JIT compiler and the analysis and parallelisation is done entirely online. The system described by the authors collects the traces independently of the Jikes RVM trace recorder that they build on, and qualifies traces into two distinct types - *execution traces* which are the instructions executed, and *memory traces*, which are traces of memory accesses. These memory traces are used as a means of checking for dependencies between traces. The authors also present a cost model for determining whether parallelisation is worthwhile, based on measuring execution time of traces and assuming that future execution time is the same.

2.7 AJITPar Project and Adaptive Skeleton Library

The Adaptive Just-In-Time Parallelisation (AJITPar) project [3][5] aims to investigate a novel approach to deliver *portable parallel performance* for programs with irregular parallelism across a range of architectures. The approach proposed combines declarative parallelism with Just In Time (JIT) compilation, dynamic scheduling, and dynamic transformation. The project aims to investigate the performance portability potential of an *Adaptive Skeletons* library (*ASL*) based on task graphs, and an associated parallel execution framework that dynamically schedules and adaptively transforms the task graphs.

ASL expresses common patterns of parallelism as a standard set of algorithmic skeletons [25], with associated transformations. Dynamic transformations, in particular, rely on the ability to dynamically compile code, which is the primary reason for basing the framework on a JIT compiler. Moreover, a trace-based JIT compiler can estimate task granularity by dynamic

profiling and/or dynamic trace cost analysis, and these can be exploited by the dynamic scheduler. A trace-based JIT-compiled functional language was chosen as functional programs are easy to transform; dynamic compilation allows a wider range of transformations including ones depending on runtime information; and trace-based JIT compilers build intermediate data structure (traces) that may be costed.

2.7.1 *ASL* Prototype Implementation

This section outlines some key design decisions and the current implementation status.

The current prototype executes task-parallel computations on shared- or distributed-memory architectures using TCP-based message passing. It implements dynamic scheduling and monitors task runtimes and communication overheads. The current system is a fairly conventional distributed-memory parallel functional language implementation; a more detailed discussion can be found in [5].

The default prototype extracts task costs using dynamic trace cost analysis based on the model in Chapter 3.

System Architecture

A high-level overview of the system is shown in Figure 2.2. The runtime environment consists of a central master and multiple workers; each being a separate OS process, possibly on different hosts. The master runs a standard Racket VM, the workers run Pycket (this architecture was chosen due to limitations of the network stack on Pycket). The master maintains the current task graph and schedules enabled task groups to idle workers. Each worker executes task groups, one task at a time, and returns the results to the master. Upon receiving results the master updates the task graph, which may unblock previously blocked task groups.

The master and workers behave much like actors, i.e. they do not share state, are single threaded and communicate by sending messages over TCP connections. In part, these design choices are born out of the restrictions of Pycket, which does not (yet) support concurrency. However, they also simplify the implementation of workers, which execute a simple receive-eval-send loop. Nonetheless, there are drawbacks compared to a shared-memory design:

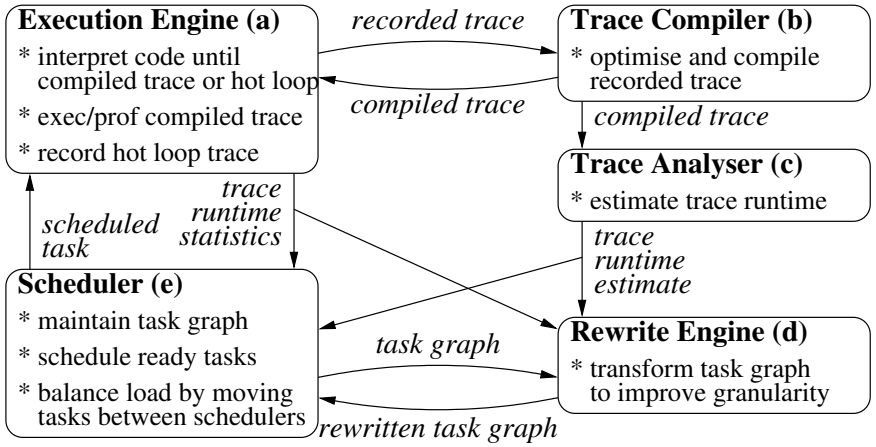

Figure 2.2: Diagram of the *ASL* runtime system. The "Trace Analyser" component is the focus of this thesis

- TCP-based message passing can add significant latency, particularly for large messages.
- All messages need to be serialised by the sender side and deserialised by the receiver, which can cause significant overhead for large messages. (In fact, Section 2.7.1 demonstrates that serialisation dominates the cost of message passing.)

The decision to adopt a centralised scheduler rather than distributed work stealing was taken with transformations in mind. It is difficult for a distributed scheduler to have a global overview of current system load and performance, as it would require for each local scheduler to continually communicate load information to every other instance, requiring a complex implementation and resulting in significant overhead [86]. This makes it difficult to decide when and how to transform skeletons.

Closures, Tasks and Serialisation

In contrast to Racket, Pycket currently expects a fixed program at startup and cannot (yet) load code dynamically. To provide the code mobility required in a distributed system, *ASL* resorts to *explicit closures*, which are essentially static global function pointers, similar to closures in distributed Haskell DSLs like CloudHaskell [87] and HdpH [45]. Tasks and task groups are layered on top, linking closures to input and output futures. Thus, evaluating a task amounts to reading its input futures, evaluating the closure and writing the result to its output future.

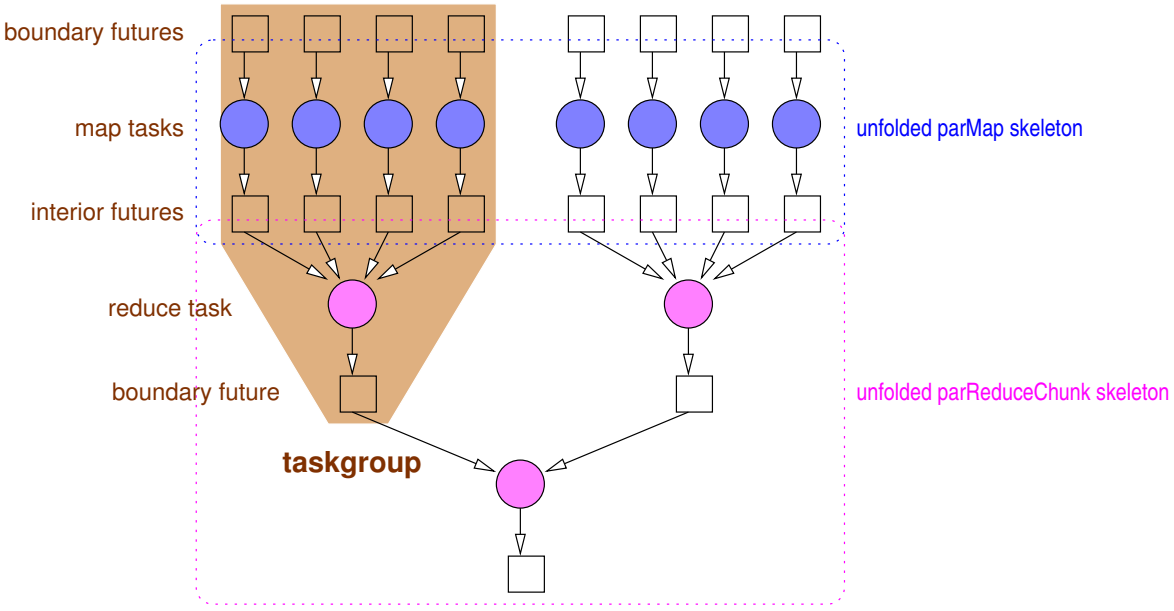

Figure 2.3: Diagram of the *ASL* task graph

Task groups (see below) and futures (results) must be serialised to byte strings that can be transmitted over TCP sockets. Racket offers a serialisation library for this purpose but the library does not work in Pycket. Hence *ASL* implements its own serialisation library, specifically designed for fast serialisation of tree-like data structures. *ASL* relies on data being acyclic; attempting to serialise cycles will likely result in the system live-locking.

Scheduling

The scheduler [5] runs entirely on the master, concurrently running one scheduling loop per worker. This loop runs until the program terminates. Each loop picks a group of tasks (with the task group size informed by the task grouping system), serialises it and sends it to a worker. The loop then waits for results to be sent back.

Task Grouping

The original vision for *ASL* was to dynamically apply transformations to the skeleton code itself; the current prototype does not do this: transformations are applied in the form of *task grouping*. Simply put, task grouping means that rather than execute each task in parallel, tasks are herded together into groups, and each group is scheduled, distributed and executed in parallel.

When a group is scheduled, the *ASL* prototype uses the cost model from Chapter 5 to cost a small sample of the tasks in that group, the costings of which are sent back to the master node with the task results. The grouping transformation engine feeds this information, along with the current number of tasks in a group, into a linear regression engine. The regression engine attempts to regress this data to an ideal task size i.e. the group size multiplied by the average cost of a task in a group should average out to this ideal number. As program execution proceeds, the regression engine is constantly working and the group sizes are continually adjusted. This ideal task group granularity is preconfigured in the prototype to be 100ms. The work in Chapter 5 is concerned with using cost modelling to improve on this ideal granularity.

2.7.2 Similar Projects

There have been previous attempts to optimise parallel programs by refactoring or transforming code. Examples include the *HWSkel* project [88, 89], *PaRTE* [90, 91, 76], SkePU [92] and the ParaForm refactorings for parallel Haskell [93]. All of these approaches use offline profiling and static transformations, and are not applied to JIT-compiled languages. In contrast ASL explores the potential of dynamic profiling and dynamic transformation of JIT-compiled code.

The Hera VM [94], is similar to *ASL* in that it uses a JIT compiled parallel language, but has significant differences to *ASL*. Hera VM is a method, rather than tracing, JIT compiler. It does not use cost models. Finally, it is specialised for the heterogeneous Cell CPU rather than cross platform performance portability.

Paraphrase Refactoring Tool for Erlang

One of these similar approaches is described in detail and it is the most similar tool to *ASL*, namely the *Paraphrase Refactoring Tool for Erlang* (PaRTE) PaRTE [90, 91, 76] is a tool which aims to allow users to discover opportunities for parallelism in Erlang programs or opportunities to improve the performance of existing parallel code. Users are presented with options for refactorings and their associated projected performance improvements; refactorings can then be performed at the touch of a button. Refactoring opportunities are discovered

through a combination of static analysis and profiling. Interaction with the tool is performed through a web interface and an Emacs plugin.

PaRTE's static analysis consists primarily of syntactic analysis. Firstly the tool tries to identify syntactic constructs which are amenable to refactoring. The tool focuses on list comprehensions, certain library function calls (particularly the `lists` module) and constructs similar to `map`. Simple dependency analysis is performed using control-flow and data-flow analysis. Code fragments are profiled using generated input data and the resulting timing information is plugged into cost models for each of the potential skeletons that could be introduced. These cost models combine the profiling information with other information such as the number of cores and the time to distribute and gather information in a *map* skeleton. Using this information, potential speedups are presented to the user for each possible refactoring.

Transformations are performed using a modified version of the wrangler Erlang refactoring tool. A set of conditional rewrite rules are used for applying skeleton refactorings based on AST matches.

Comparison with *ASL* As they set out to achieve similar goals and both involve transformations to enable or improve parallelism, *ASL* and PaRTE seem superficially similar. However, there are a number of major and minor differences between them. The first significant difference is that PaRTE performs analysis on high level syntax while *ASL*'s cost analysis is performed on low-level intermediate representation. PaRTE also relies heavily on offline profiling for cost analysis, plugging actual timing information into its cost models, while AJITPar's models calculate cost using static analysis applied to dynamic information - the trace instructions. This is a direct result of the greatest difference between the two tools — PaRTE's analysis and transformations are performed offline and ahead of time, while *ASL*'s are performed dynamically and in soft real time. Another difference is that PaRTE can semi-automatically introduce parallelism to a sequential program, while AJITPar requires that a program be written in the AJITPar parallel DSL. There are also other differences in explicitness — PaRTE requires user intervention before refactorings are performed, while AJITPar performs the transformations automatically at runtime.

Chapter 3

Costing JIT Traces

3.1 Introduction

Tracing JIT compilation generates units of compilation that are easy to analyse and are known to execute frequently. The AJITPar project investigates whether the information in JIT traces can be used to dynamically transform programs for a specific parallel architecture. Hence a lightweight cost model is required for JIT traces.

This chapter presents the design and implementation of a system for extracting JIT trace information from the Pycket JIT compiler (section 3.3). Section 3.4 describes three increasingly parametric cost models for Pycket traces and determines the best values for the cost model. Section 3.5 evaluates the effectiveness of the cost models for predicting the relative costs of transformed programs.

3.2 Pycket Trace Structure

The cost models described in this chapter are built on the particular structure of Pycket/PyPy traces. A detailed discussion of the low-level structure of these traces follows.

A JIT *trace* consists of a series of instructions recorded by the interpreter, and a trace becomes *hot* if the number of jumps back to the start of the trace (or loop) is higher than a given threshold, indicating that the trace may be executed frequently and is worth compiling.

Other important concepts in Pycket traces include *guards*: assertions which cause execution to leave the trace when they fail; *bridges*: that are traces starting at a guard that fails often enough; and *trace graphs*: representing sets of traces. The nodes of a trace graph are entry points (of loops or bridges), labels, guards, and jump instructions. The edges of a trace graph are directed and indicate control flow. Note that control flow can diverge only at guards and merge only at labels or entry points. A *trace fragment* is a part of a trace starting at a label and ending at a jump, at a guard with a bridge attached, or at another label, with no label in between.

The listing in Figure 3.1 shows a Racket program incrementing an accumulator in a doubly nested loop, executing the outer loop 10^5 times and the inner loop 10^5 times for each iteration of the outer loop, thus counting to 10^{10}.

```
(define numb1 100000)
(define numb2 100000)

(define (inner iter acc)
  (if (> iter numb2)
      acc
      (inner (+ iter 1) (+ acc 1))))

(define (outer iter acc)
  (if (> iter numb1)
      acc
      (outer (+ iter 1) (inner 0 acc))))

(outer 0 0)
```

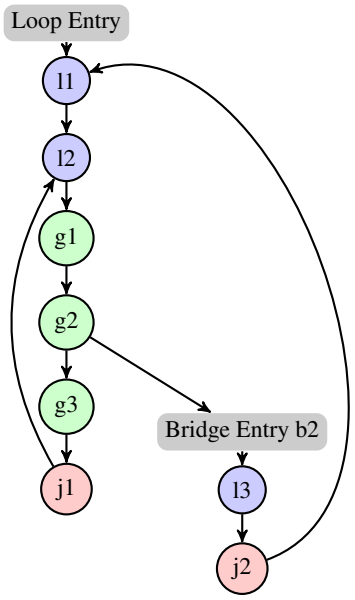

Figure 3.1: Doubly nested loop in Racket and corresponding Pycket trace graph.

Figure 3.1 also shows the trace graph produced by Pycket. The nodes represent instructions which are pertinent to the flow of control through the loop. In the graph, labels are represented by l nodes, g nodes represent guards and j nodes represent jump instructions. The inner loop (which becomes hot first) corresponds to the path from l2 to j1, and the outer loop corresponds to the bridge. The JIT compiler unrolls loops once to optimise loop invariant code, producing the path from l1 to l2.

The trace graph is a convenient representation to read off the trace fragments. In this example, there are the following four fragments: l1 to l2, l2 to g2, l2 to j1, and l3 to j2. Trace

```
label(i7, i13, p1, descr=TargetToken(4321534144))
debug_merge_point(0, 0, '(let ([if_0 (> iter numb2)]) ...)')
guard_not_invalidated(descr=<Guard0x10196a1e0>) [i13, i7, p1]
debug_merge_point(0, 0, '(> iter numb2)')
i14 = int_gt(i7, 100000)
guard_false(i14, descr=<Guard0x10196a170>) [i13, i7, p1]
debug_merge_point(0, 0, '(if if_0 acc ...)')
debug_merge_point(0, 0, '(let ([AppRand0_0 ...] ...) ...)')
debug_merge_point(0, 0, '(+ iter 1)')
i15 = int_add(i7, 1)
debug_merge_point(0, 0, '(+ acc 1)')
i16 = int_add_ovf(i13, 1)
guard_no_overflow(descr=<Guard0x10196a100>) [i16, i15, i13, i7, p1]
debug_merge_point(0, 0, '(inner AppRand0_0 AppRand1_0)')
debug_merge_point(0, 0, '(let ([if_0 (> iter numb2)]) ...)')
jump(i15, i16, p1, descr=TargetToken(4321534144))
```

Figure 3.2: Trace fragment l2 to j1.

fragments can overlap: for instance, l2 to j1 overlaps l2 to g2.

Figure 3.2 shows a sample trace fragment, l2 to j1, corresponding to the inner loop. Besides debug instructions, the fragment consists of 3 arithmetic-logical instructions and 3 guards (only the second of which fails often enough to have a bridge attached).

The label at the start brings into scope 3 variables: the loop counter i7, the accumulator i13, and a pointer p1 (which plays no role in this fragment). The jump at the end transfers control back to the start and also copies the updated loop counter and accumulator i15 and i16 to i7 and i13, respectively.

3.3 Language Infrastructure

3.3.1 Runtime Access to Traces and Counters

This section builds on the concepts of *traces, fragments* and *guards* introduced in Section 3.2. The RPython tool chain provides language developers with a rich set of APIs to interact with its generic JIT engine. Among these APIs are a number of callbacks that can intercept intermediate representations of a trace, either immediately after recording, or after optimisation.

In debug mode RPython can instrument traces with counters, recording how often control

reaches an entry point or label. RPython provides means to inspect the values of these counters at runtime. *ASL* uses this feature to derive estimates of the cost of whole loop nests from the cost and frequency of their constituent trace fragments.

The JIT compiler counts the number of times a label is reached but we are more interested in counting the execution of traces. Unfortunately, full traces as gathered by our system cannot be simply counted, as guards can fail and jumps can target any label. Fortunately, we can work out the trace fragment execution count due to the fact that there is a one-to-one correspondence between guards and their bridges. Essentially, the frequency of a fragment ℓ to g is the frequency of the bridge attached to guard g. Trace fragments are the largest discrete part of traces we can accurately count. The frequency of a fragment starting at ℓ and not ending in a guard is the frequency of label ℓ minus the frequency of all shorter trace fragments starting at ℓ. Tables 3.1 and 3.2 demonstrate this on the trace fragments of the nested loop example in fig. 3.1. The first two columns show the JIT counters, the remaining three columns show the frequency of the four trace fragments, and how they are derived from the counters. Note that not all counters reach the values one would expect from the loop bounds. This is because counting only starts once code has been compiled; iterations in the warm-up phase of the JIT compiler are lost. The hotness threshold, determined from the Pycket source code, is 131 loop iterations.

JIT counter	JIT count
n_{l1}	100,001
n_{l2}	10,000,098,957
n_{b2}	99,801
n_{l3}	99,800

Table 3.1: JIT counters and counts for program in Figure 3.1.

fragment	frequency expression	frequency
l1 to l2	n_{l1}	100,001
l2 to g2	n_{b2}	99,801
l2 to j1	$n_{l2} - n_{b2}$	9,999,999,156
l3 to j2	n_{l3}	99,800

Table 3.2: JIT counters and trace fragment frequencies for program in Figure 3.1.

Class	Example Instructions
$debug$	debug_merge_point
$numeric$	int_add_ovf
$guards$	guard_true
$alloc$	new_with_vtable
$array$	arraylen_gc
$object$	getfield_gc

Table 3.3: RPython JIT Instruction Classes

3.3.2 An analysis of Pycket JIT instructions

It is useful to classify the RPython JIT instructions into different sets, conceptually. Ignoring debug instructions ($debug$ is the name of this set), the set of all instructions is named all. The set all can then be divided into two theoretical sets: high cost instructions ($high$) e.g. memory allocation, and low cost instructions (low) e.g. numerical instructions.

These two sets can be further decomposed into five intuitive sets. The classes are object read and write instructions $object$, guards $guards$, numerical instructions $numeric$, memory allocation instructions $alloc$ and array instructions $array$. These classes are described in Table 3.3. Jump instructions are ignored, since there is only ever one in a trace. External calls are excluded as foreign function code is not represented within the trace, and thus cannot be costed.

A histogram of JIT operations, taken from traces generated by all the cross-implementation benchmarks [95] and shown in Figure 3.3, shows that overall these traces are also dominated by instructions from the $guards$, $objects$ and $numeric$ classes.

3.4 JIT-based Cost Models

The traces produced by Pycket during JIT compilation provide excellent information for cost analysis. The linear control flow makes traces easy to analyse, and the fact that traces are only generated for sufficiently “hot” code focuses cost analysis on the most frequently executed code paths. In this section, we define several cost models based on trace information collected from Pycket.

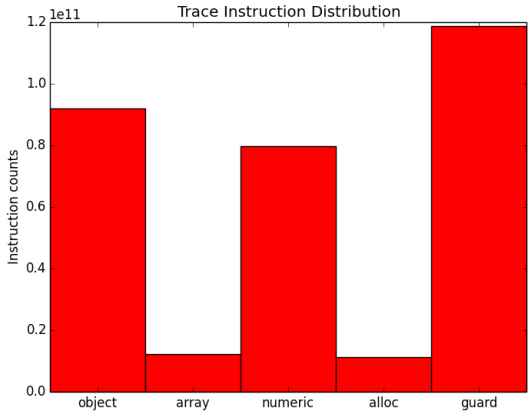

Figure 3.3: Most common instructions in cross-implementation Pycket benchmarks

3.4.1 Trace Cost Models

Let *Tr* be an arbitrary trace of length n, that is, $Tr = op_1 \dots op_n$ is a sequence of instructions op_i. A trace cost model γ is a function mapping *Tr* to its *predicted trace cost* $\gamma(Tr)$, where $\gamma(Tr)$ is a dimensionless number, (ideally) proportional to the time to execute *Tr*. Since the runtime of *Tr* may depend on the hardware architecture, the trace cost model is specific to a particular architecture.

Null Cost Model (CM_0)

The simplest possible trace cost model assigns the same cost to each trace, regardless of its length and the instructions contained. The purpose of this *null cost model*, which is formally defined by Equation (3.1), is to serve as a baseline to compare the accuracy of other cost models against. Using this model to calculate the cost for whole programs (Section 3.4.2) can be considered roughly equivalent to using a loop counting control-flow analysis for estimating the execution time of a program. Note that the null cost model is architecture independent.

$$\gamma(Tr) = 1 \tag{3.1}$$

Counting Cost Model (CM_C)

A slightly more sophisticated trace cost model declares the cost of a trace to be its length, counting the number of instructions (ignoring debug instructions, which are not executed at

runtime). This *counting cost model* is defined by Equation (3.2) and is architecture independent.

$$\gamma(\mathit{Tr}) = \sum_{i=1}^{n} \begin{cases} 0, & \text{if } \mathit{op}_i \in debug \\ 1, & \text{otherwise} \end{cases} \tag{3.2}$$

Weighted Cost Model (CM_W)

Certain types of instructions are likely to have greater execution time, for example memory accesses may be orders of magnitude slower than register accesses. A more intricate cost model can be obtained by applying a weighting factor to each of the instruction classes described in Section 3.3.2. Equation (3.3) shows the definition of this *weighted cost model*, parameterised by abstract weights a, b, c, d and e.

$$\gamma(\mathit{Tr}) = \sum_{i=1}^{n} \begin{cases} 0, & \text{if } \mathit{op}_i \in debug \\ a, & \text{if } \mathit{op}_i \in array \\ b, & \text{if } \mathit{op}_i \in numeric \\ c, & \text{if } \mathit{op}_i \in alloc \\ d, & \text{if } \mathit{op}_i \in guard \\ e, & \text{if } \mathit{op}_i \in object \end{cases} \tag{3.3}$$

The accuracy of the model depends on the concrete weights, and their choice depends on the actual architecture. Section 3.4.3 demonstrates how to obtain concrete weights for a reasonably accurate model.

3.4.2 Whole Program Cost Models

Let P be a program. During an execution of P, the JIT compiler generates m distinct trace fragments Tr_j and m associated trace counters n_j.

Given a (null, counting or weighted) trace cost model γ, the (null, counting or weighted) *cost* $\Gamma(P)$ of P is defined by summing up the cost of all traces, each weighted by their execution

frequency; see Equation (3.4) for a formal definition.

$$\Gamma(P) = \sum_{j=1}^{m} n_j \, \gamma(\mathit{Tr}_j) \tag{3.4}$$

Note that Γ is not a *predictive* cost model, as its definition relies on traces and trace counters, and the latter are only available after the execution of a program. However, Γ can still be useful for predicting the cost of transformations, as demonstrated in Section 3.5.

3.4.3 Calibrating Weights for CM_W

To use the abstract weighted cost model CM_W (Section 3.4.1), it is necessary to find concrete values for the weight parameters $a, \ldots, e$ in Equation (3.3). Ideally, program cost $\Gamma(P)$ is proportional to program runtime $t(P)$. That is, ideally there exists $k > 0$ such that Equation (3.5) holds for all programs P.

$$\Gamma(P) = k \, t(P) \tag{3.5}$$

Given sufficiently many programs and sufficiently varied program inputs, we can use Equation (3.5) to calibrate the weights of CM_W for a given architecture by linear regression, as detailed later in this section.

Benchmarks

For the purpose of calibrating weights we use 41 programs from the standard Pycket benchmark suite *pycket-bench* [96] and the Racket *Programming Languages Benchmark Game* suite [97]. The programs used are a subset of the full suite of 121 as programs that result in failing benchmark runs or which contain calls to foreign functions are omitted. Foreign function calls are removed as it is unlikely that any two foreign function calls are doing the same thing or take the same time.

For each program, we record the execution time, averaging over 10 runs. All traces and the values of all trace counters are recorded; since all benchmarks are deterministic traces and trace counters do not vary between runs.

The Pycket version used for these experiments is revision e56ba66d71 of the `trace-analysis` branch of our custom fork [98], built with Racket version 6.1 and revision 79009 of the RPython toolchain. The experiments are run on a 2.0 GHz Xeon server with 64 GB of RAM running Ubuntu 14.04.

Linear Regression

Picking an arbitrary value for k, e.g. $k = 1$, we derive the following relation from Equations (3.5) and (3.4).

$$t(P_l) = \Gamma(P_l) + \epsilon_l = \sum_{j=1}^{m_l} n_{lj}\, \gamma(\mathit{Tr}_{lj}) + \epsilon_l \tag{3.6}$$

P_l is the lth benchmark program, generating m_l traces Tr_{lj} and trace counters n_{lj}, $t(P_l)$ is the observed average runtime of P_l, and ϵ_l is the error term. Equation (3.6) becomes a model for linear regression by expanding γ according to its definition (3.3), which turns the right-hand side into an expression linear in the five unknown weights $a, \ldots, e$.

Weights are implicitly constrained to be non-negative, as negative weights would suggest that corresponding instructions take negative time to execute, which is physically impossible. To honour the non-negativity constraint, weights are estimated by non-negative least squares linear regression.

$$\gamma(\mathit{Tr}) = \sum_{i=1}^{k} \begin{cases} 4.884 \times 10^{-4}, & \text{if } \mathit{op}_i \in \mathit{numeric} \\ 4.797 \times 10^{-3}, & \text{if } \mathit{op}_i \in \mathit{alloc} \\ 4.623 \times 10^{-4}, & \text{if } \mathit{op}_i \in \mathit{guard} \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

$$\gamma(\mathit{Tr}) = \sum_{i=1}^{k} \begin{cases} 2.987 \times 10^{-4}, & \text{if } \mathit{op}_i \in \mathit{numeric} \\ 1.574 \times 10^{-4}, & \text{if } \mathit{op}_i \in \mathit{array} \\ 4.122 \times 10^{-4}, & \text{if } \mathit{op}_i \in \mathit{object} \\ 2.919 \times 10^{-4}, & \text{if } \mathit{op}_i \in \mathit{guard} \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

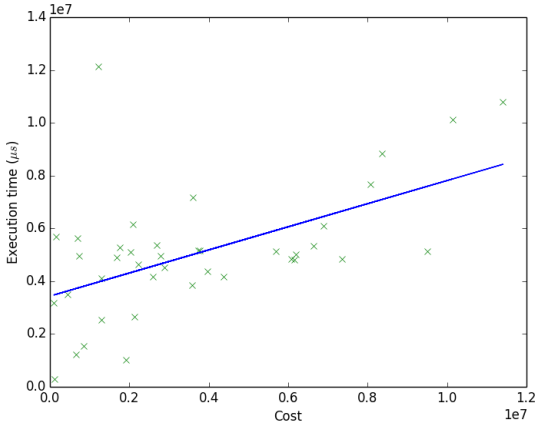

Figure 3.4: Execution time vs cost for CM_W determined using linear regression

Equation (3.7) shows the resulting weighted cost model for the GPG cluster node, while Equation (3.8) shows the same for the FATA machine. Equation (3.7) only attributes non-zero cost to allocation, numeric instructions and guards, implying that object and array access instructions have negligible cost. However, Equation (3.8) attributes zero cost to allocation, and broadly similar costs to the other, non-debug instructions. We suspect that the difference between the two architectures is due to improved memory bandwidth on the FATA node.

The regression fit for the cost model in Equation (3.7) is shown in Figure 3.4. The fit is obviously linear but rather coarse, indicating that CM_W is not a very accurate model. The square of the residual (R^2) value for this fit is 0.29.

There is one egregious outlier (the `trav2` benchmark – a tree traversal program) with no obvious explanation. The benchmark does have larger trace outputs than others — 5.5 MB compared to an average of 1.7 MB, but this is not enough to account for the discrepancy. The time spent tracing doesn't account for this outlier either: other benchmarks spend much greater time tracing and compiling as a proportion of execution time without producing such divergence. We note that linear regression fits for CM_C and CM_0 are visibly worse than the fit for CM_W, which implies that their accuracy is lower than CM_W.

3.5 Costing Transformations

The main purpose of a cost model in the AJITPar project is to enable the selection and parameterisation of appropriate dynamic transformations. This section describes the transfor-

mations and explores how accurately the cost models predict the execution time of programs before and after transformation.

3.5.1 Skeleton Transforms

In AJITPar parallel programs are expressed by composing *algorithmic skeletons* [25] from an Adaptive Skeletons (AS) library [99].

Adaptive skeletons are based on a standard set of algorithmic skeletons for specifying task-based parallelism within Racket. The AS framework expands skeletons to task graphs and schedules tasks to workers; expansion and scheduling happen at runtime to support tasks with irregular granularity. The AS framework piggy-backs on Pycket to analyse the cost of tasks as they are executed. The cost information is used both to guide the dynamic task scheduler as well as a skeleton transformation engine. The latter *adapts* the task granularity of the running program to suit the current architecture by rewriting skeletons according to a standard set of equations [99].

A number of different skeleton types are used in *ASL*. The basic types of skeletons are *parallel map*, *parallel reduce* and *divide and conquer*. The actual versions of the skeletons in AJITPar are *tuneable*, in that they are parameterised with a number that specifies the granularity of the parallelism in some way. The definitions of some of these tuneable skeletons, `parMapChunk`, `parMapStride` and `parDivconqThresh`, are shown in Figure 3.5, specified in a Haskell-style pseudocode[1]. Code which uses these skeletons can be transformed by modifying the first argument which serves as a tuning parameter; we will use τ to denote this tuning parameter.

A notable abscence is the *pipeline* skeleton, which allows other skeletons to be composed together in such a way that the second skeleton can begin parallel compuation before the first has completed i.e. a pair of parallel maps pipelined so that the second map stars work on the first element of the result vector of the first map while the first map is still computing. Due to the lack of full worker to worker communication without synchronising with a central master, a pipeline skeleton is currently not feasible in *ASL*. *ASL* currently lacks full worker to worker communication. Implementing the pipeline skeleton in such a system would require all

[1]Extended with a primitive spawn, where expressions of the form spawn f x create a new task computing the function application f x.

workers to synchronise with the master, the overhead of which would wipe out any parallel performance gain. As such, the pipeline skeleton is not currently feasible in *ASL*.

The *ASL* system aims to transform skeletons such that the resulting tasks are of optimal granularity, i.e. execute in the range of 10-100ms. This target task granularity is based on previous scaling experiments of *ASL* infrastructure [100]. An optimal task granularity should result in optimal parallel scaling.

To this end, the system monitors the runtime of tasks and computes their cost as they complete, following Equation (3.4). If the system sees too many tasks fall outwith the optimal granularity range, it will attempt to transform the skeleton that generated the tasks. In the simplest case this is done by changing the tuning parameter τ as follows.

Let t_0 and γ_0 be the observed average runtime and cost of tasks generated by the skeleton's current tuning parameter τ_0. The system computes $k = t_0/\gamma_0$ and picks a target granularity t_1 (in the range 10 to 100 milliseconds) and corresponding target cost $\gamma_1 = t_1/k$. Then the system picks the new tuning parameter τ_1 such that the cost ratio γ_1/γ_0 and the tuning ratio τ_1/τ_0 are related by the skeleton's cost derivative.

The *cost derivative* is the rate of change of cost γ with respect to the change in the tuning parameter τ. For example, the cost derivative for the parMapChunk skeleton is the constant function 1 because doubling the chunk size τ doubles the cost of tasks. In contrast, the derivative for parMapStride is the function $1/x$ because doubling the stride width τ halves the cost of individual tasks. In general, the cost derivative is specific to the skeleton but independent of benchmark application and architecture.

Underlying this method of tuning τ is the assumption that the time/cost ratio k is independent of τ. The rest of this section will empirically demonstrate that this is indeed the case as long as task granularity is not too small.

3.5.2 Experiments

The suitability of the cost models for predicting the effect of applying transforms on execution time is evaluated. A cost model will be considered sufficiently accurate if the ratio k of execution time to predicted cost is constant across different τ values for each program.

```
-- map skeletons
parMap :: (a → b) → [a] → [b]
parMap f []     = []
parMap f (x:xs) = spawn f x : parMap f xs

parMapChunk :: Int → (a → b) → [a] → [b]
parMapChunk k f xs = concat $ parMap (map f) $ chunk k xs

parMapStride :: Int → (a → b) → [a] → [b]
parMapStride k f xs = concat $ transpose $ parMap (map f)
                      $ transpose $ chunk k xs

-- divide and conquer skeletons
parDivconq :: (a → [a]) → ([b] → b) → (a → b) → a → b
parDivconq div comb conq x =
  case div x of
    [] → spawn conq x
    ys → spawn comb (map (parDivconq div comb conq) ys)

parDivconqThresh :: (a → Bool) → (a → [a]) → ([b] → b)
                    → (a → b) → a → b
parDivconqThresh thresh div comb conq x
  = if thresh x
    then spawn (divconq div comb conq) x
    else case div x of
           [] → spawn conq x
           ys → comb (map (parDivconqThresh p div comb conq) ys)

-- signatures of auxiliary functions
chunk     :: Int → [a] → [[a]]
map :: (a → b) → [a] → [b]
concat    :: [[a]] → [a]
divconq :: (a → [a]) → ([b] → b) → (a → b) → a → b
transpose :: [[a]] → [[a]]
```

Figure 3.5: AJITPar base skeletons and tunable skeletons.

Benchmark	Input	Skeleton(s)
Matrix multiplication	1000x1000 matrices	parMapChunk
SumEuler	$[1 \ldots 4000]$	parMapChunk; parMapStride
Fibonacci	42	parDivconqThresh
k-means	sample data	parMapChunk
Mandelbrot	6000x6000	parMapChunk

Table 3.4: Benchmarks with their input and applied skeletons

Benchmarks and transforms

The benchmarks used in these experiments are shown in Table 3.4, and the sources of the benchmarks are available at [101]. For most benchmarks it is obvious what tasks compute, e.g. in the case of matrix multiplication a chunk of rows of the result matrix. k-means is a special case, its tasks do not compute a clustering but classify a chunk of the input data according to the current centroids; this is the parallel part of each iteration of the standard cluster refinement algorithm. The input data for k-means consists of 1024000 data points of dimension 1024, to be grouped into 5 clusters. The experiments are carried out on the same hardware and software platforms as in Section 3.4.3.

Experimental Design

The benchmarks represent the sequential code executed by a worker during the execution of a single task. Each benchmark is run with a variety of different values for the tuning parameter τ. For example, Fibonacci is run with threshold values of 15, 16, 17, 18, 21, 24, 27, 30, etc. Since Pycket does not yet support snapshots of the trace counter file, each run is performed twice; once with warmup code only and then again with the warmup code and the task that is to be measured. The difference in trace counters between the two runs accurately reflects to the trace counters of the task[2].

Mandelbrot and SumEuler are *irregular* benchmarks, that is, work is distributed non-uniformly, making some tasks harder than others. To investigate the accuracy of the cost model in the presence of irregular parallelism, we repeat the Mandelbrot and chunked SumEuler experiments with different chunks.

[2]Unless the JIT was not warmed up sufficiently.

Results

The graphs of time/cost ratio k against tunable paramater τ for each benchmark and cost model can be found in Figures 3.6 and 3.11. The rightmost point on each graph represents the τ equivalent to one worker, and thus the untransformed version of that code; moving rightwards along the x-axis corresponds to increasingly coarse-grained tasks.

Figure 3.12 shows the plot of k (for cost model CM_W) against τ for each of three different chunks of Mandelbrot, showing how irregularity affects the prediction. Table 3.5 shows the stable values of time/cost ratio k to which the benchmarks converge; the table also shows the range of values that k can take and a "minimum" task granularity (Section 3.5.3).

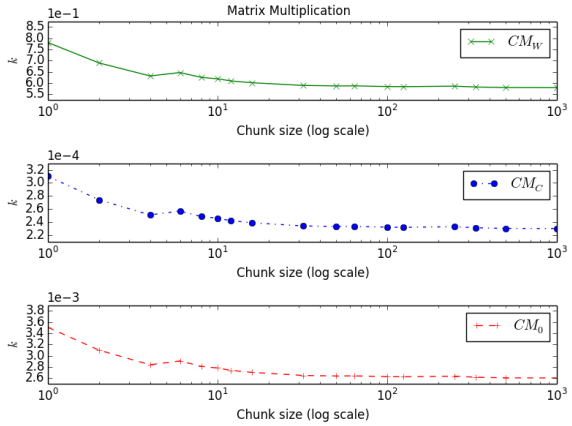

Figure 3.6: k vs τ for Matrix multiplication benchmark

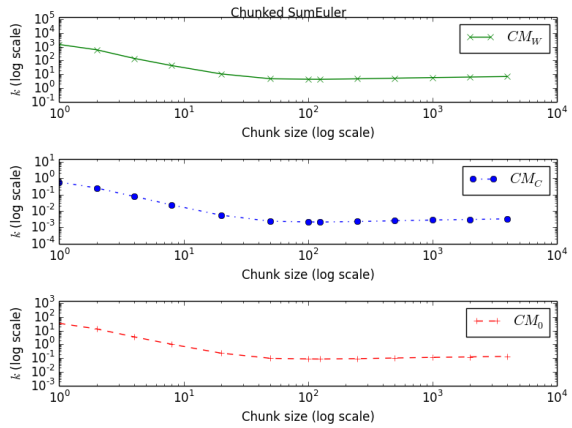

Figure 3.7: k vs τ for irregular chunked SumEuler benchmark

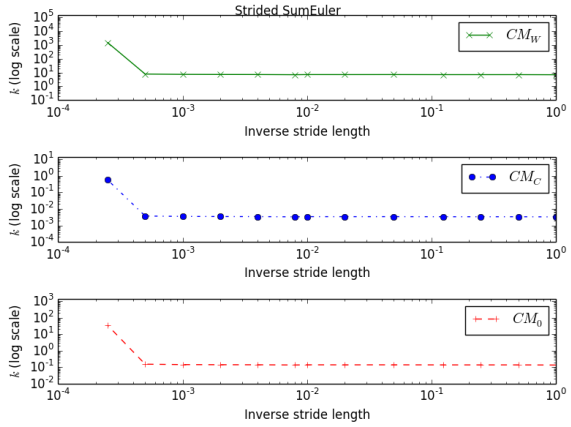

Figure 3.8: k vs τ for strided SumEuler benchmark

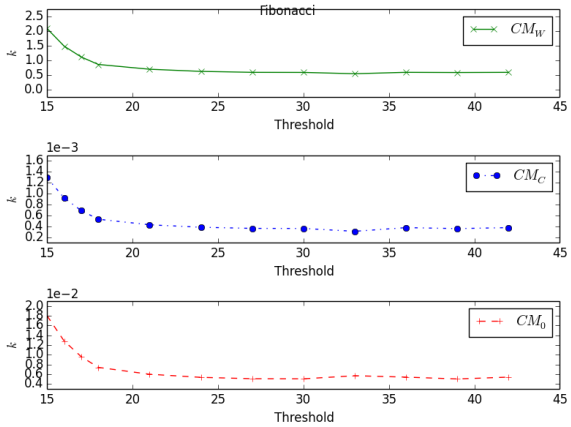

Figure 3.9: k vs τ for Fibonacci benchmark

3.5.3 Discussion

The overall shape of graphs in Figures 3.6 and 3.11 is the same for all benchmarks and cost models: The time/cost ratio k starts out high (on the left) and falls at first as task granularity increases, then stabilises. The value of k the graphs stabilise at depends on the benchmark and on the cost model; for CM_W the stable k values are listed in Table 3.5. By design of CM_W these values cluster around 1 though none of them is particularly close to 1, indicating that CM_W is not particularly accurate for any of the benchmarks, over- or under-estimating the actual execution time by a factor of 2 to 7. This is expected given the coarseness of the fit of CM_W shown in the previous section (Figure 3.4). Similar graphs are produced by running the same benchmarks on the FATA platform, with CM_W parameterised for that platform. These figures can be found in Appendix A.3

One difference between the graphs is the range over which k varies as task granularity in-

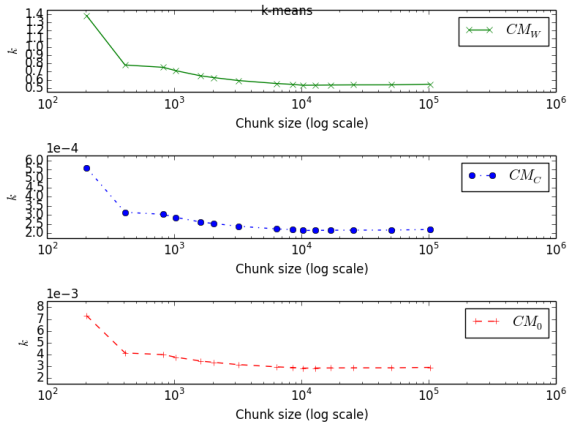

Figure 3.10: k vs τ for k-means benchmark

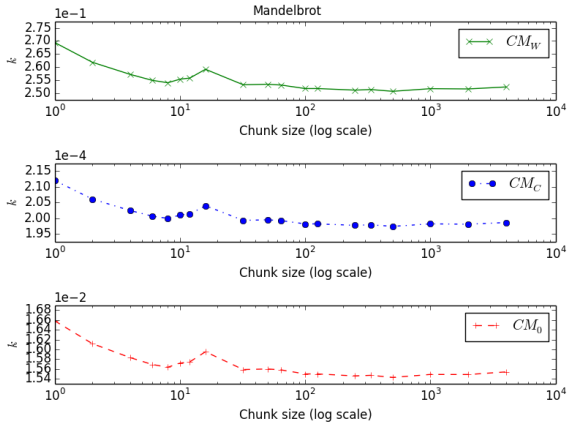

Figure 3.11: k vs τ for Mandelbrot benchmark

creases; this range is listed in Table 3.5. For the SumEuler benchmarks, and to a lesser extent for Fibonacci, this range is large. This correlates with very low granularities (on the order of tens of microseconds) for the smallest tasks. Once the granularity crosses a certain threshold, around 100 to 300 μs as listed in Table 3.5, the value of k stabilises. This suggests that the cost models are particularly inaccurate for small tasks, possibly due to the fact that smaller tasks run through fewer traces, but do become more accurate as task size increases. In particular, the cost models are reasonably accurate for tasks in the target granularity range of 10 to 100 milliseconds.

For matrix multiplication and Mandelbrot the range of k listed in Table 3.5 is small. For k-means the range would also be small (around 0.3) if the unusually high k for the smallest task granularity were disregarded as an outlier.[3] This correlates with minimum task granularities

[3]The experiment data suggest this outlier is caused by insufficient JIT warmup though we do not yet understand why.

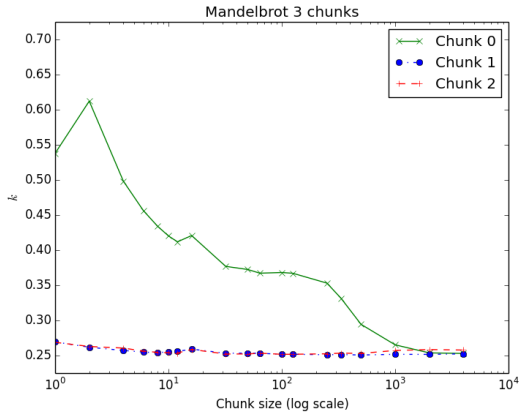

Figure 3.12: k vs τ for Mandelbrot benchmark (CM_W) comparing 3 chunks

Benchmark	stable k	range of k	mintask granularity for stable k
Matrix Multiplication	0.579	0.201	$< 11400\ \mu s$
Strided SumEuler	6.87	1450	$306\ \mu s$
Chunked SumEuler	4.31	1460	$129\ \mu s$
Fibonacci	0.542	1.55	$294\ \mu s$
k-means	0.535	0.847	$< 12100\ \mu s$
Mandelbrot	0.251	0.0187	$< 117000\ \mu s$

Table 3.5: Stable k values for each benchmark (cost model CM_W)

that are quite high (10 to 120 milliseconds); in fact, these granularities are already in the target range. Thus, for these benchmarks the cost models are reasonably accurate over the whole range of the tuning parameter τ.

Another source of inaccuracy for cost prediction, besides ultra-low task granularity, is irregularity. The chunked SumEuler and Mandelbrot benchmarks do exhibit irregular parallelism. In the case of SumEuler, chunks at the lower end of the interval give rise to smaller tasks than chunks at the upper end, and in the case of Mandelbrot, chunks at the top and bottom of the image produce smaller tasks than chunks in the middle. The graphs in Figures 3.7 and 3.11 show plots of k for chunks in the middle of the interval or image rather than the average over all chunks, in an attempt to account for the effect of irregularity. Figure 3.12 contrasts the time/cost ratio k of a chunk at the top of the image (Chunk 0) with two chunks in the middle. The k for Chunk 0 is markedly different from the other two and not stable, though the graphs do converge as granularity increases, which correlates with the fact that irregularity decreases as chunk size increases. We note that while the moderate irregularity

of Mandelbrot causes some loss of accuracy, it is not too bad: the ratio between the most extreme k of Chunk 0 and the stable value of k for Mandelbrot is less than a factor of 3. In contrast, the ratio between task runtimes for Chunk 0 and average task runtimes for Mandelbrot is a factor of more than 10. Thus, the cost models are somewhat able to smooth the inaccuracies in prediction that are caused by irregular task sizes.

Finally, on the evidence presented here, it does look like all three cost models are equally well suited to predicting the cost of transformations. While this is the case for simple transformations that only change the value of a single tuning parameter τ, this need no longer be the case when trying to cost a chain of two transformations. In future work, we aim to systematically predict the cost of chains of transformations of skeleton expressions comprising multiple skeletons, e.g. a parallel map followed by a parallel reduce. It is expected that in these cases there will be a bigger difference between the set of traces pre- and post-transformation than currently seen. Hence the actual content of the traces should matter more, and cost model CM_W to beat the other two on accuracy of prediction.

3.6 Performance Overhead

The performance overhead of applying Γ at runtime was measured between 0.25% and 18% for the set of Racket *Programming Languages Benchmark Game* benchmarks. The level of overhead depends on the specific program being run, but generally decreases with increased program execution time, as shown in Figure 3.13. This is unsurprising as a longer execution time will amortise the cost of running the cost model.

3.7 Discussion

This chapter has discussed the design and implementation of a system for extracting JIT trace information from the Pycket JIT compiler (Section 3.3). Three lightweight cost models for JIT traces, ranging from the extremely simple loop counting model CM_0 to the relatively simple instruction counting model CM_C to the architecture-specific weighted model CM_W, have been defined. To automatically determine appropriate weights for CM_W, linear regression over the Pycket benchmark suite has been performed on two different hardware

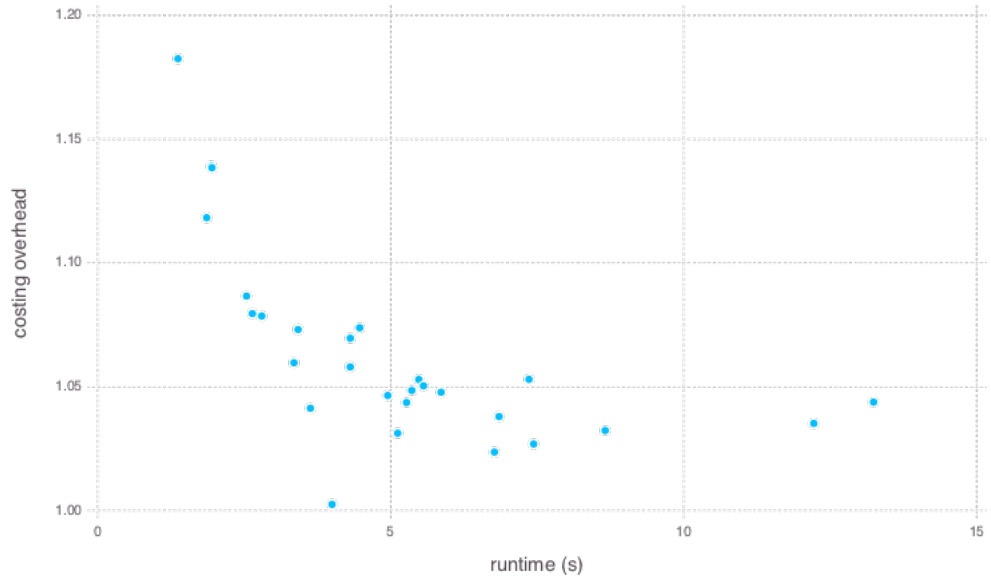

Figure 3.13: Costing overhead vs program execution time for a set of 28 benchmarks

platforms (Section 3.4). All three cost models have been used to compare the relative cost of tasks generated by six skeleton-based benchmarks pre- and post-transformation, where the skeleton transformations are induced by changing a skeleton-specific tuning parameter. The effect of these transformations on task runtime can be predicted accurately using the cost models, once the task granularity rises above a threshold (Section 3.5).

We have demonstrated that even the simplest, architecture-independent cost model described in this paper allows us to accurately predict the effect of simple transformations on task runtime.

Chapter 4

Communications Cost Modelling

ASL needs to make decisions about scheduling tasks and transforming task graphs to improve parallel performance. It is speculated that performance can be improved if, in addition to measuring the computation cost as in the previous chapter, *ASL* measures the communication cost, i.e. the costs of serialising, deserialising and transmitting the data encoded in tasks. As the cost model is intended to be executed during program warm up, it must be very cheap to execute, and hence is a simple linear model.

This chapter describes the development of increasingly detailed abstract cost models for communication overhead. These abstract models are parameterised by applying linear regression to performance measurements of serialisation, deserialisation and network transmission of Racket and Pycket data types. The final cost model is validated and along with its *additive property* i.e that cost model instances for primitive data types can be combined to accurately predict cost models for compound data types.

The work described in this chapter is believed to constitute the first dynamic recursive type-driven cost model for serialisation, deserialisation and network communication.

This chapter is structured as follows. Section 4.1 outlines the requirements of a communications cost model for *ASL*. Section 4.2 describes the design and development of the communications cost model K, and the derivation of weightings for simple, known types. Section 4.3 discusses an additivity property of a typed communications cost model, and describes the validation of this property for K. Section 4.4 details the integration of K into *ASL*. Finally, Section 4.5 outlines the cross validation of K.

4.1 Requirements of a Communication Cost Model

A communications model for *ASL* must meet a number of requirements to be useful. First, since it must be applied dynamically, the model must be simple. The model must also account for the *ASL* architecture. Second, the model must account for the fact that the master and worker nodes use different language platforms - Racket and Pycket. The model must also account for the different communication patterns on both the master and worker. Finally, the model must be able to account for arbitrary data structures. It is not possible to predict in advance what types a user may use.

4.2 Designing a Cost Model

This section describes the design and development of the communications cost model K.

Note the figures in this section refer to those on the GPG platform only; those for FATA can be found in Appendix B.2

4.2.1 Original Design

Initially, it was speculated that the communication cost, K_{naive}, would be determined by network communication and serialisation. Both these properties were modelled on the size of the serialised data in bytes, l. This definition of the cost model is given in equation 4.1. The other symbols are defined as follows: n is a weighting factor for the network communications, s is a weighting factor for serialization/deserialisation is some constant. Note that a term for a constant overhead was originally considered for inclusion in this and later equations. Please see Appendix B.1 for a discussion of this.

$$K_{naive} = l(n + s) \tag{4.1}$$

4.2.2 Hardware and Software Environment

The benchmarks are run on the same hardware and software platforms as in Chapter 3 GPG, a Beowulf cluster of 16 2.0GHz Intel Xeon servers with 64 GB of RAM running Ubuntu 14.04;

Type	Description
bstr	byte string
flmatrix	vector of flvectors
float	float
flvector	floating point vector
int	integer
list	linked-list
string	string
vecbytes	vector of bytes
vecstring	vector of strings
vector	fixed length array of ints
vector2	vector of vectors of ints

Table 4.1: Type name explanations

and FATA a 32-core 2.6Ghz Xeon server with 64GB of RAM. Revision `d45e79919f` of branch `runtime_trace_analysis` of an *ASL*Pycket fork [98] is used as the Pycket platform and Racket 6.5 as the Racket platform. Branch `comms` of *ASL* was used.

4.2.3 Initial Experiments

A pair of simple experiments attempted to calibrate the model i.e. to determine values for n and s from Equation (4.1).

The experiment measures serialisation time for the different data types in Table 4.1 by serialising each data type 5 times and calculating the average time to serialise that data type. This is repeated 2000 times for each data type, with a varying random input parameter which determines the size/shape of the data structure.

The network send time experiment measures the time taken for a worker node to send byte-strings of random sizes to a netcat node in listen mode. This is repeated 200 times, with a different random sized byte-string each time. In both cases, linear regression is used to identify the parameter values. The sample sizes chosen differ between the serialisation and network send time experiments in order to enable the experiments to complete in practical time. The experiments are repeated for both Racket and Pycket workers. In the case of the GPG platform, transmission is between two nodes over gigabit Ethernet. On the FATA platform, the transmission is on a single node using the loopback interface.

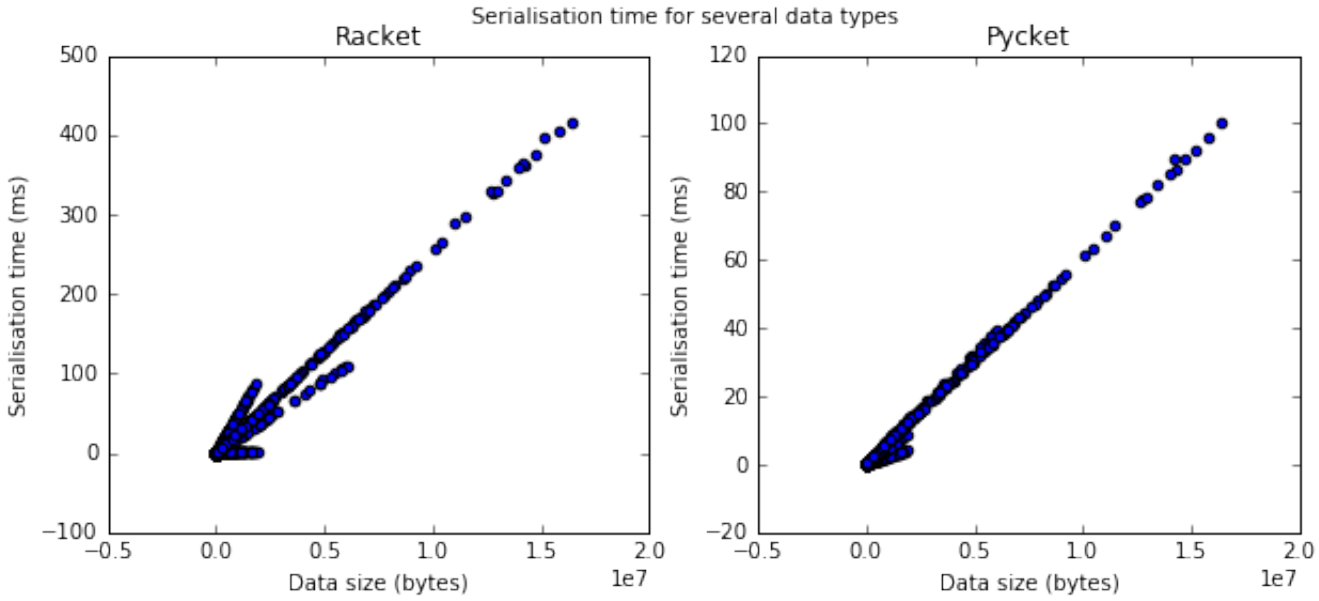

Figure 4.1: Serialisation results

Results

The results for serialisation are found in Figure 4.1, and the results for network communication are seen in Figure 4.2.

Figure 4.1 shows clear multimodal distribution for both Racket and Pycket as there are several different lines extending from a single point. This makes it impossible to calculate a weighting, as there is no linear relation. Separating the data by the data type shows a set of obvious linear relationship, suggesting that there is a type dependency in the cost model. Figures 4.3 and 4.4 contain an example selection of the graphs: three examples showing linear behaviour, and one showing the two phase behaviour (the remaining graphs can be found in Appendix B). The two-phase behaviour for strings in Racket is probably because larger strings change memory allocation performance.

The values for the gradients and y-intercepts for the serialisation costs of each data type for each architecture can be found in Tables 4.4 and 4.5. The gradients of these fits are used as the parameter values in the cost model for serialisation. The values for each system are generally of the same order of magnitude.

Figure 4.2 shows clear linear relationships between the network send time and the size in bytes of the data structure being sent. Tables 4.2 and 4.3 present the gradients for the network communications graphs for each architecture. These values are generally an order of

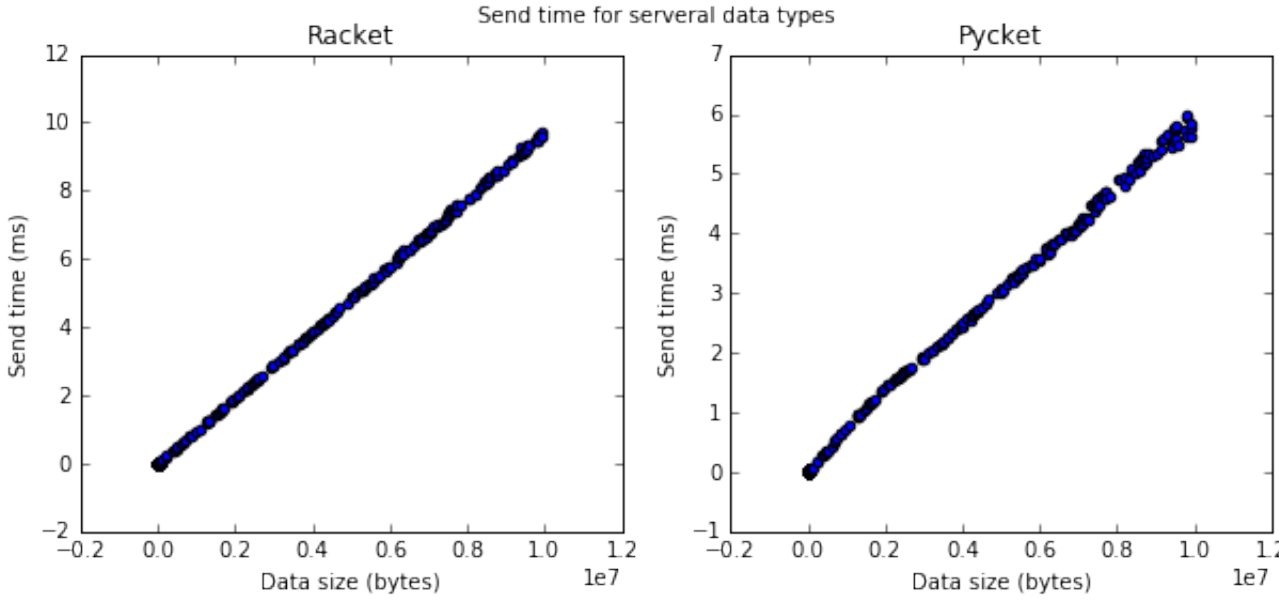

Figure 4.2: Network send time results, Intel Xeon 2.0GHz, 1Gb Ethernet

Worker	Gradient $(\mathrm{ms/byte})$
Racket	9.6897×10^{-7}
Pycket	6.0258×10^{-7}

Table 4.2: Network Send Gradients (GPG - node to node 1Gb Ethernet)

Worker	Gradient $(\mathrm{ms/byte})$
Racket	2.6812×10^{-7}
Pycket	5.4321×10^{-7}

Table 4.3: Network Send Gradients (GPG - node to node 1Gb Ethernet)

Type	Gradient (racket) $(\mathrm{ms/byte})$	Gradient (pycket) $(\mathrm{ms/byte})$
bstr	1.6225×10^{-6}	1.9898×10^{-6}
flmatrix	1.8194×10^{-5}	6.2774×10^{-6}
flvector	1.7494×10^{-5}	6.2076×10^{-6}
list	4.6706×10^{-5}	1.5795×10^{-5}
string	4.7687×10^{-5}	4.7224×10^{-6}
vecbytes	1.7206×10^{-6}	2.0796×10^{-6}
vecstring	4.7920×10^{-5}	4.7379×10^{-6}
vector	2.4959×10^{-5}	5.8001×10^{-6}
vector2	2.5349×10^{-5}	5.7468×10^{-6}

Table 4.4: Serialisation parameters (GPG node)

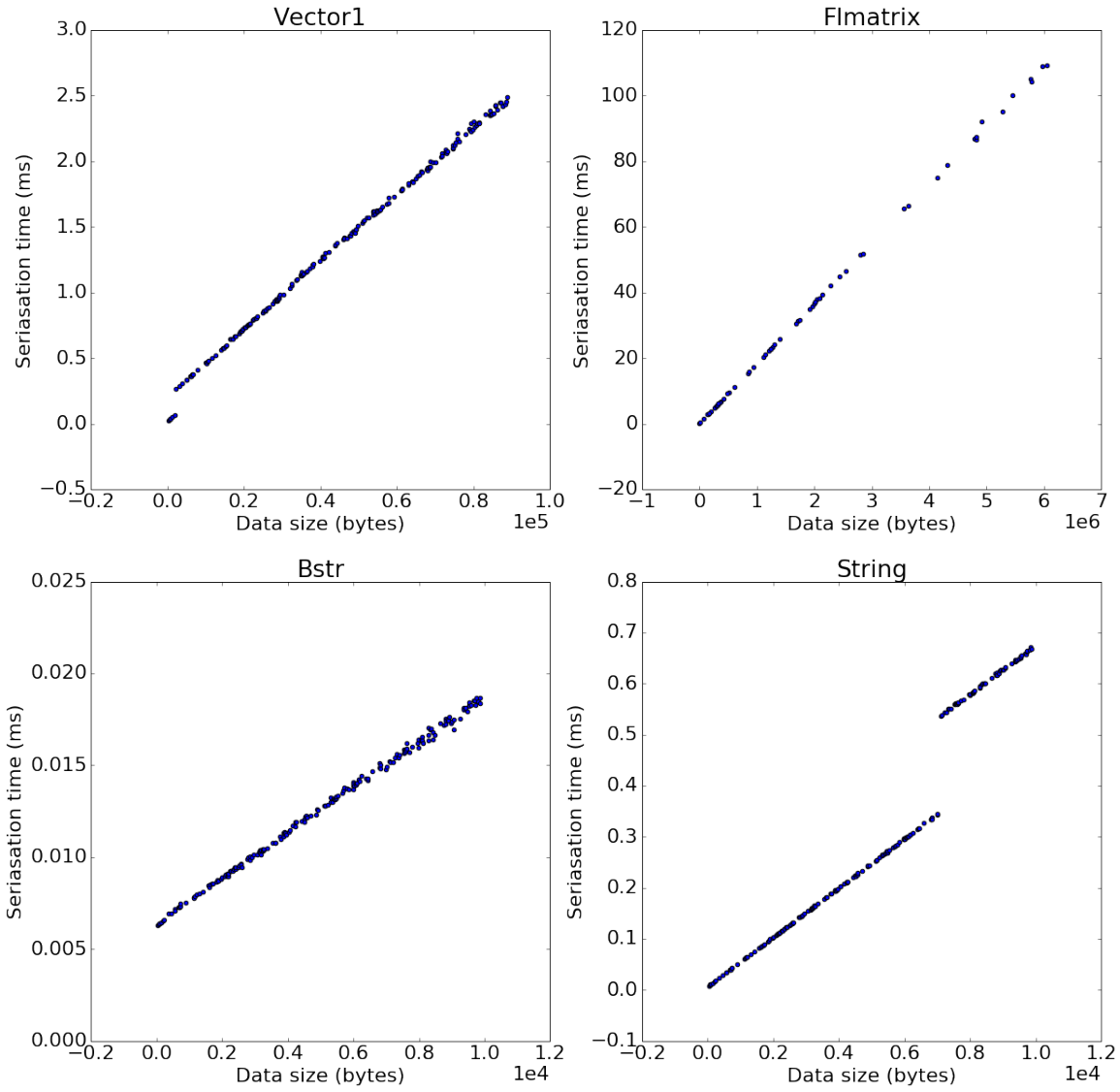

Figure 4.3: Serialisation time against Data Size (separated by type) (Racket; GPG)

Type	Gradient (racket) (ms/byte)	Gradient (pycket) (ms/byte)
bstr	2.3484×10^{-6}	2.2438×10^{-6}
flmatrix	3.2836×10^{-5}	8.6881×10^{-6}
flvector	1.4561×10^{-5}	6.0715×10^{-6}
list	3.5171×10^{-5}	1.2415×10^{-5}
string	3.8189×10^{-5}	4.7601×10^{-6}
vecbytes	9.6551×10^{-5}	4.9820×10^{-5}
vecstring	0.0002	5.5505×10^{-5}
vector	1.9805×10^{-5}	8.0888×10^{-6}
vector2	0.0001	6.8407×10^{-5}

Table 4.5: Serialisation parameters (FATA node)

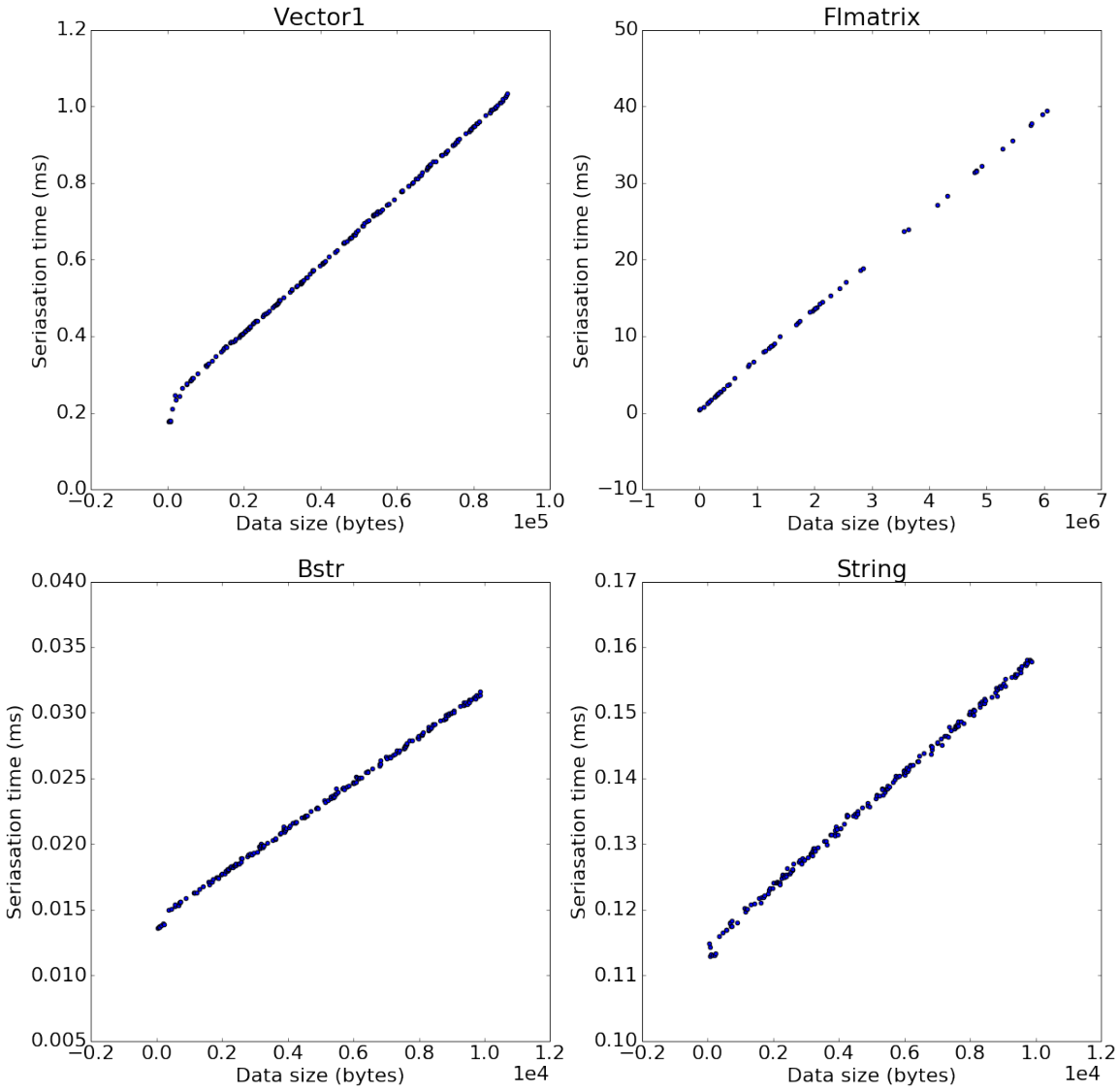

Figure 4.4: Serialisation Time against Data Size (separated by type) (Pycket; GPG)

magnitude smaller than the corresponding values for serialisation.

Discussion

The results in section 4.2.3 expose some deficiencies in K_{naive}. The multimodal distributions in Figure 4.1 show that a single value for s cannot be used as a predictor for a model. However, Figures 4.3 and 4.4 show that if we separate serialisation time by type, linear models for a type-parameterised s can be produced. It is noted that serialisation is significantly more costly than network transmission. The network transmission cost on the FATA node is less than on the GPG node, but not by as much as expected.

4.2.4 Type-indexed Model

Extending K_{naive} to account for types solves some of its problems. In equation 4.2, the cost model function K_{typed} now has a parameter x (instances of this model exist for Racket and Pycket). The symbol x represents the data structure being serialised/transmitted and is of type t. The serialisation cost s is now parameterised in terms of t, and the values of s for different t can be found in equation 4.3. Formerly the scalar length of the data in bytes, l is now a function which returns the size of x in bytes. The values for s for each t for the GPG and FATA architectures are found in Tables 4.4 and 4.5 respectively. These values are obtained from the experiments performed in Section 4.2.3, by separating the results by type. GPG and FATA produce broadly similar parameter values, but those produced on FATA are generally more costly.

$$K_{typed}(x :: t) = l(x)(n + s(t)) \tag{4.2}$$

$$s(t) = \begin{cases} 0.312 \times 10^{-5} & \text{if } t = numeric \\ 5 \times 10^{-4} & \text{if } t = list \\ ... & \\ ... & \\ etc. & \end{cases} \tag{4.3}$$

4.2.5 Type-indexed Bidirectional Communication Model

K_{typed} (Equation 4.2) is more refined than K_{naive}, but does not account for two-way communication —the Racket master node sending task data to the Pycket workers, and the Pycket workers send the results of those tasks back to the Racket master (Section 2.7.1).

A new model, $K_{bidirectional}$, takes into account that the task data is serialised on the racket master and then transmitted. The return value is serialised on pycket and then transmitted. The user program could return any data type so the equation needs to be parameterised in terms of the data structure sent, x, and received, y.

$$K_{bidirectional}(x :: t, y :: u) = K_r(x) + K_p(y) \tag{4.4}$$

$$K_r(x :: t) = l(x)(n_r + s_r(t)) \tag{4.5}$$

$$K_p(x :: t) = l(x)(n_p + s_p(t)) \tag{4.6}$$

$K_{bidirectional}$, is defined as the sum of the Racket cost of sending x, $K_r(x)$, and the Pycket cost of receiving y, $K_p(y)$.

$K_r(x)$ and $K_p(y)$ are defined in Equations (4.5) and (4.6) respectively. In equation 4.5, for racket, n_r is the network send parameter value and s_r is the serialisation weighting. In equation 4.6, for pycket, n_p is the network send parameter value and s_p is the serialisation weighting. A full model of the communication cost is beginning to come together.

4.2.6 Type-indexed Bidirectional Serialisation/Deserialisation Model

$K_{bidirectional}$ includes the time taken to serialise a data structure and send it, but so far it fails to say anything about what happens to the serialised data structure when it reaches the other end of the transmission. A serialised message must be deserialised on the other end of the communication channel. In *ASL*, deserialisation uses distinct algorithms from deserialisation, and have differing performance (this is borne out by the differences between results in Tables 4.4 and 4.6).

To incorporate the different serialisation and deserialisation costsK_{tbsd} (tbsd - *Type-indexed Bidirectional Serialisation/Deserialisation*), the s term is replaced with the term sd which include the deserialisation costs, Equations (4.8) and (4.9) for Racket and Pycket respectively. Each serialisation/deserialisation term is then expanded to be the sum of the appropriate serialisation and deserialisation weightings; the expansion for the Racket master node is shown in Equation (4.10) and the dual for the Pycket worker is shown in Equation (4.11).

4.2.7 Deserialisation Experiments

These experiments intend to parameterise values for Equations (4.10) and (4.11). Similarly to Section 4.2.3, measurement of the deserialisation time and linear regression are used to determine the weightings and overheads.

Methodology

This experiment measures deserialisation time for different serialised data types by deserialising the binary representation of each data type 5 times and calculating the average time to deserialise that data type. This is repeated 1000 times for each data type, with a varying random input parameter to determine the size/shape of the data structure.

Results Again, the results show that the deserialisation time is linearly proportional to the size of the structure to be deserialised in bytes. A selection of results from the twelve tested types are shown in Figures 4.5 and 4.6, showing obvious linear behaviour and some non-linear exceptions. Memory allocation behaviour is suspected to be responsible for the non-linear or multi-phase behaviour. Tables 4.6 and 4.7 contains the parameter values for each architecture. Again, the GPG and FATA platforms produce broadly similar parameters, but deserialisation on FATA is generally more costly.

$$K_{tbsd}(x :: t) = K_r(x) + K_p(x) \tag{4.7}$$

$$K_r(x :: t) = l(x)(n_r + sd_r(t)) \tag{4.8}$$

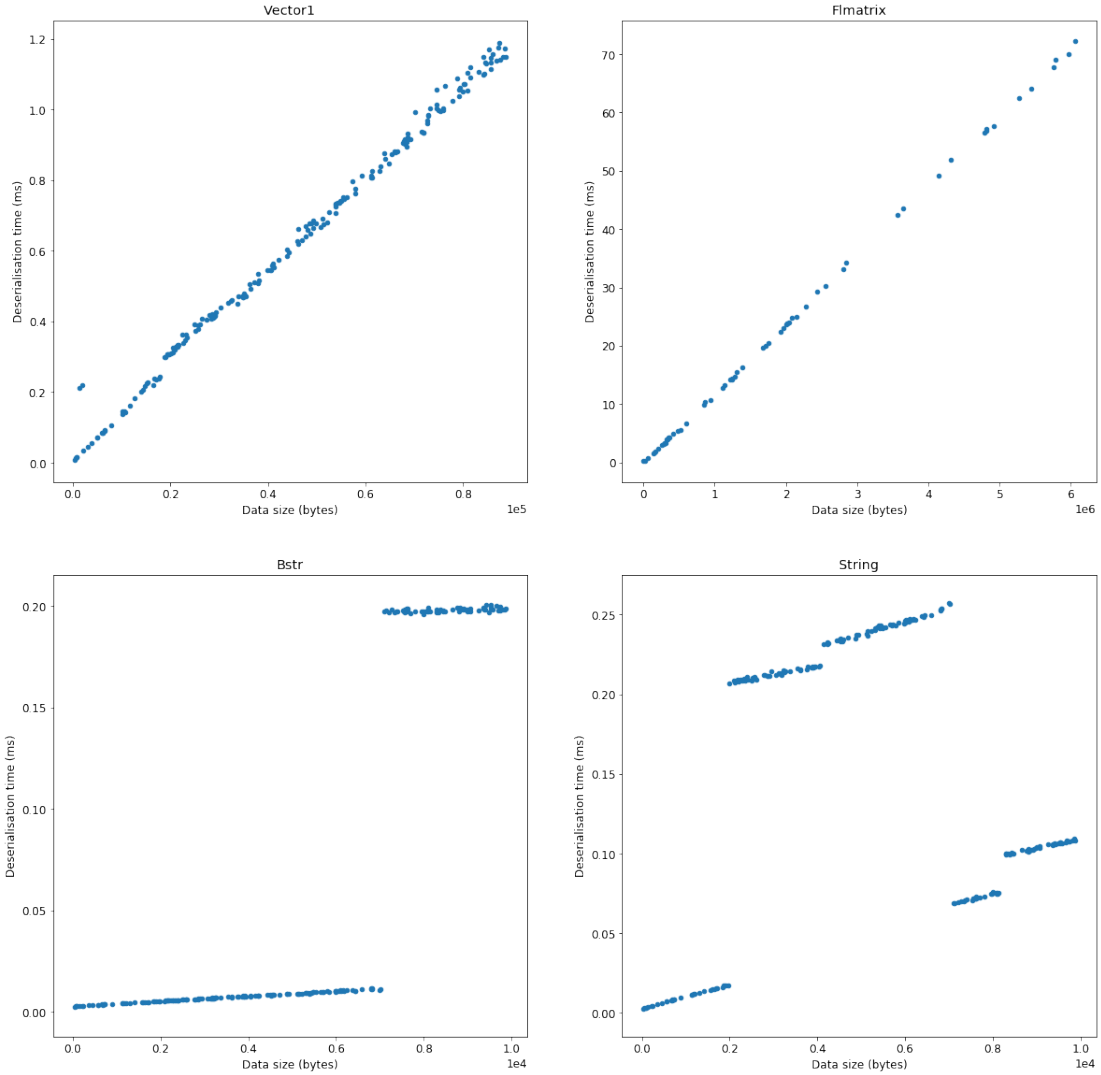

Figure 4.5: Deserialisation Time against Data Size separated by type (Racket; GPG)

Type	Gradient (Racket) (ms/byte)	Gradient (Pycket) (ms/byte)
bstr	1.4517×10^{-6}	3.7799×10^{-7}
flmatrix	1.1487×10^{-5}	3.9492×10^{-6}
flvector	1.0878×10^{-5}	3.8078×10^{-6}
list	5.8485×10^{-5}	4.1530×10^{-5}
string	6.7701×10^{-6}	1.4029×10^{-6}
vecbytes	1.6326×10^{-6}	8.5128×10^{-7}
vecstring	1.0073×10^{-5}	1.7119×10^{-6}
vector	2.4100×10^{-5}	1.0725×10^{-5}
vector2	1.5310×10^{-5}	1.0288×10^{-5}

Table 4.6: Deserialisation parameters (GPG)

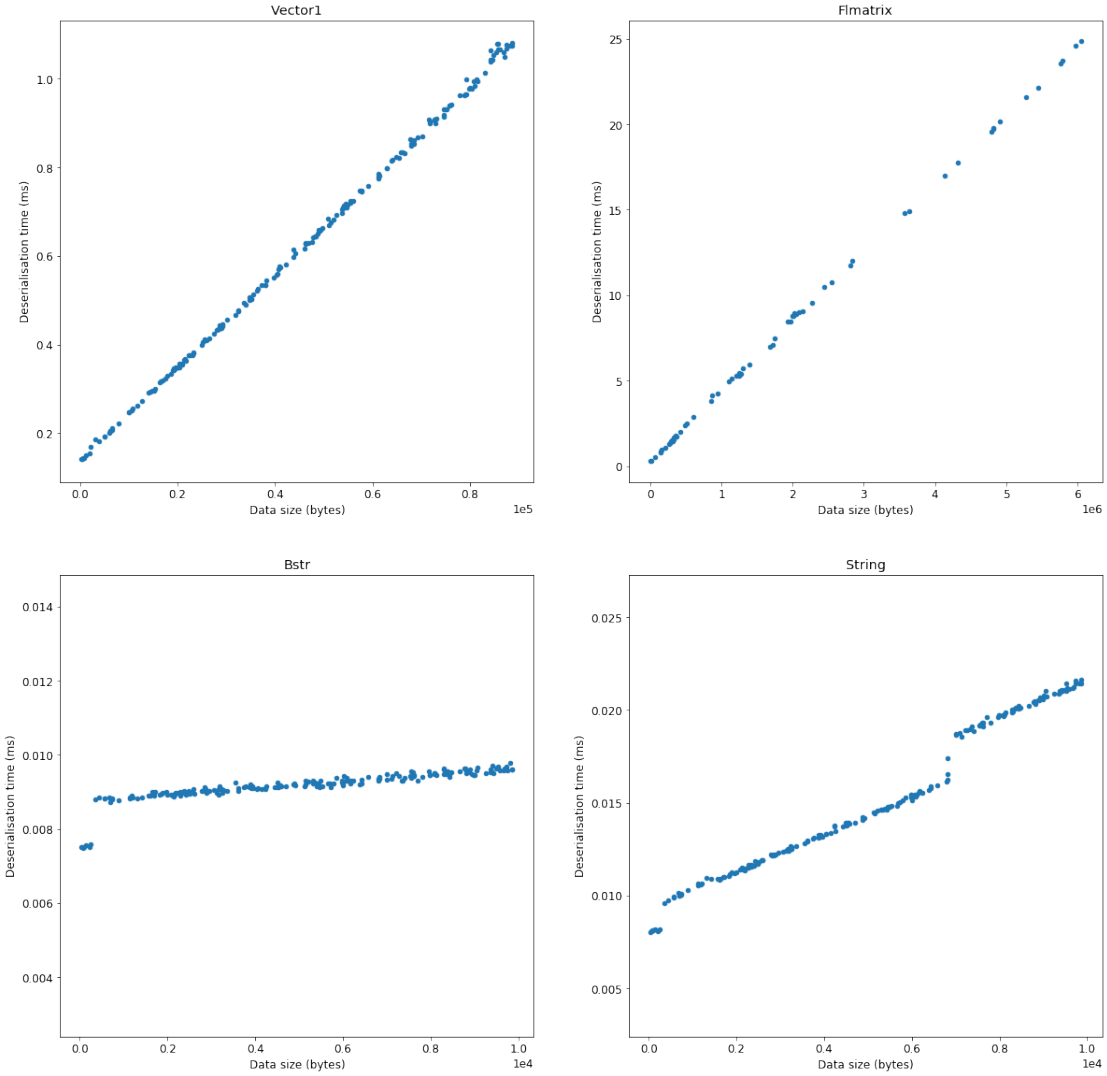

Figure 4.6: Deserialisation Time against Data Size separated by type (Pycket; GPG)

Type	Gradient (Racket) (ms/byte)	Gradient (Pycket) (ms/byte)
bstr	9.4100×10^{-7}	5.2340×10^{-7}
flmatrix	1.2263×10^{-5}	3.7345×10^{-6}
flvector	8.9018×10^{-6}	3.2911×10^{-6}
list	4.6259×10^{-5}	3.1729×10^{-5}
string	4.2442×10^{-6}	8.4670×10^{-7}
vecbytes	4.5160×10^{-5}	1.5891×10^{-5}
vecstring	7.9376×10^{-5}	2.1175×10^{-5}
vector	1.1266×10^{-5}	8.6151×10^{-6}
vector2	5.6672×10^{-5}	2.6268×10^{-5}

Table 4.7: Deserialisation parameters (FATA)

$$K_p(x :: t) = l(x)(n_r + sd_p(t)) \tag{4.9}$$

$$sd_r(t) = s_r(t) + d_p(t) \tag{4.10}$$

$$sd_p(t) = s_p(t) + d_r(t) \tag{4.11}$$

4.2.8 Discussion

The results in Section 4.2.7 show that generally, the (de)serialisation times of the primitive types are linear with respect to their data size and thus suitable for calibrating the model K_{tbsd}. It is noted that (de)serialisation dominates the model, begin an order of magnitude more costly than network transmission. K_{tbsd} meets two of the requirements specified in Section 4.1 in that it accounts for both Racket and Pycket and it accounts for the different communications patterns on the master and worker nodes. The GPG and FATA platforms produce broadly similar parameters, but deserialisation on FATA is generally more costly.

4.3 Validating an Additive Property of Cost Model

The cost model K_{tbsd} has inferred weightings for 11 simple data types, but it is possible - even likely - that the data that will be seen in a live system will consist of aggregations of these simple data types or aggregations of those aggregations. It is unfeasible to enumerate each likely possible combination of data types and test them as in section 4.2.3 and section 4.2.7. However, this is unnecessary if the cost model is additive, i.e the communication cost of an aggregate data type of another type is equal to the sum of the costs of the other type, with a weighting factor varying depending on the type of the containing type.

The required additive property is illustrated in Equation (4.12) , where the (de)serialisation cost s of an aggregate type t containing types of u is defined as a weighting factor a multiplied by the sum of (de)serialisation costs s of each u. Note that this example has a *homogeneous* aggregate type - every structure contained in t is of type u. *Heterogeneous* aggregate

types, where the member types are not all the same, are also supported in the *ASL* system, and Equation (4.12) must also hold in this situation for K to be useful.

$$s(t(u)) = a(t) \sum_{n=0}^{len(t)} s(u) \tag{4.12}$$

4.3.1 Experiments

To investigate this additive property we experiment with homogeneous and heterogeneous aggregate types, comparing the actual time taken to serialise the aggregate data structure with the cost predicted by Equation (4.12). When plotting the actual time against the predicted, if a linear relationship exists, the additive property can be said to hold, and the gradient of this linear model will be the parameter a for that type.

Note that these experiments refer to the GPG platform only.

Heterogeneous Additive Types

A vector containing different data structures of random sizes is repeatedly (de)serialised and the time taken to do so recorded.

The results for racket can be found in Figures 4.9 and 4.10 and those for pycket can be found in Figures 4.7 and 4.8. Clear linear relationships can be seen for both serialisation and deserialisation, strongly suggesting that the additive property holds for vectors of heterogeneous types.

Homogeneous Additive Types

This experiment is conducted similarly to that in section 4.3.1, except that the data structures tested exclusively consist of increasingly nested vectors of integers and increasingly nested lists of integers both up to a maximum dimension of 5 considered in addition to vectors. The results show a linear relationships between the predicted cost and the actual serialisation time, strengthening the conclusion that the additive property holds for both vectors and lists of heterogeneous types.

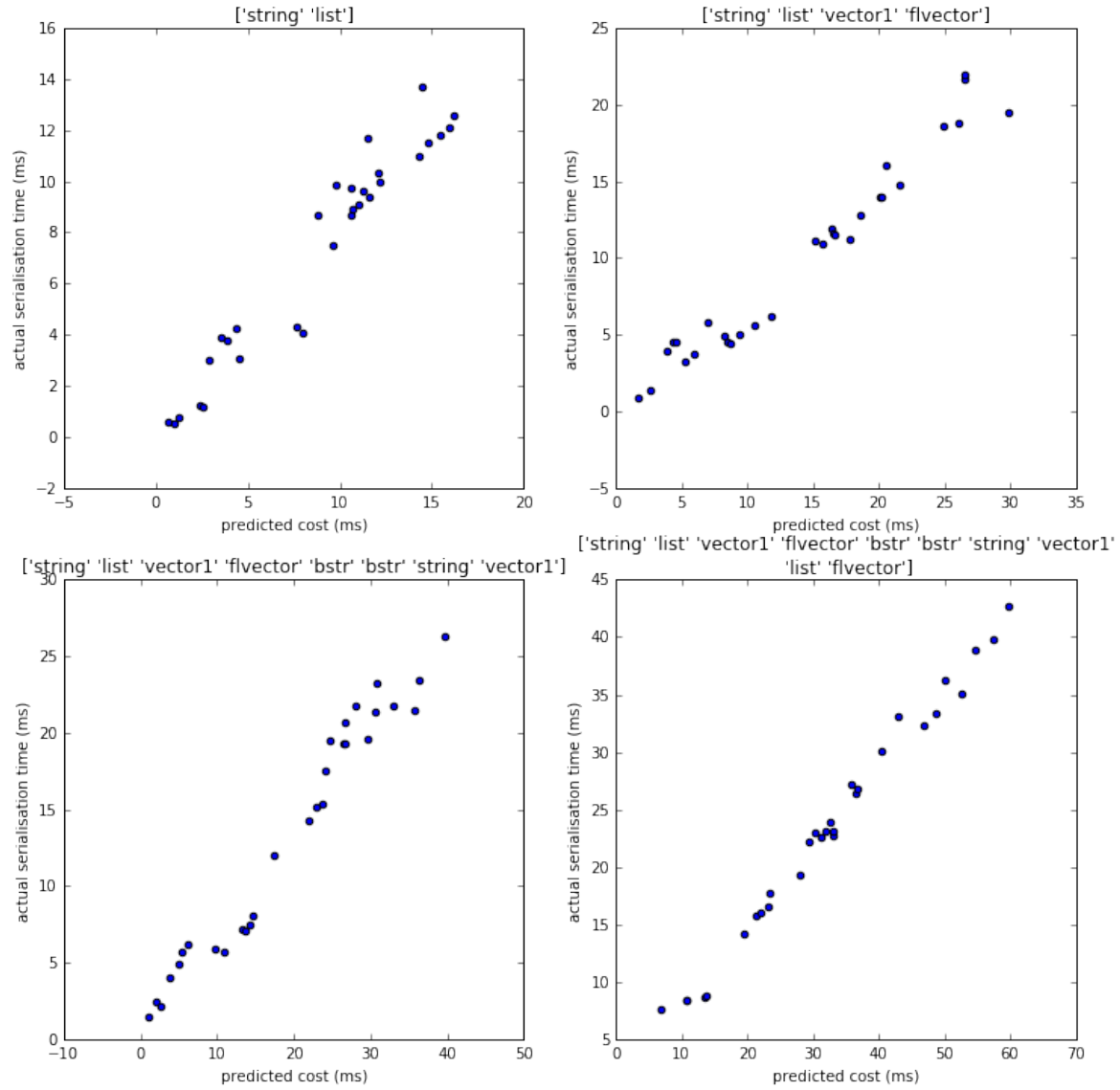

Figure 4.7: Actual serialisation time vs predicted serialisation time for heterogeneous tuples (Pycket)

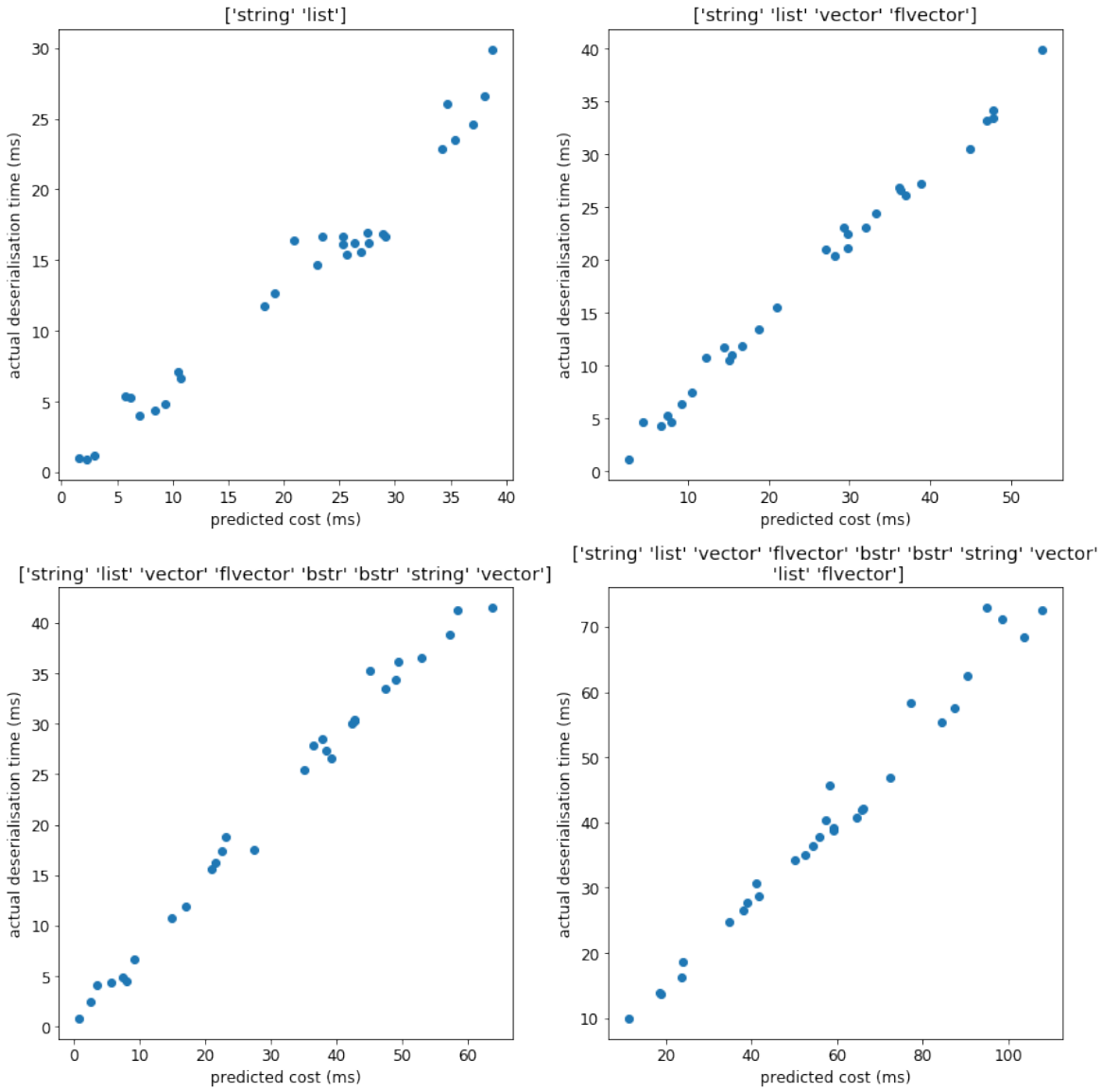

Figure 4.8: Actual deserialisation time vs predicted deserialisation time for heterogeneous tuples (Pycket)

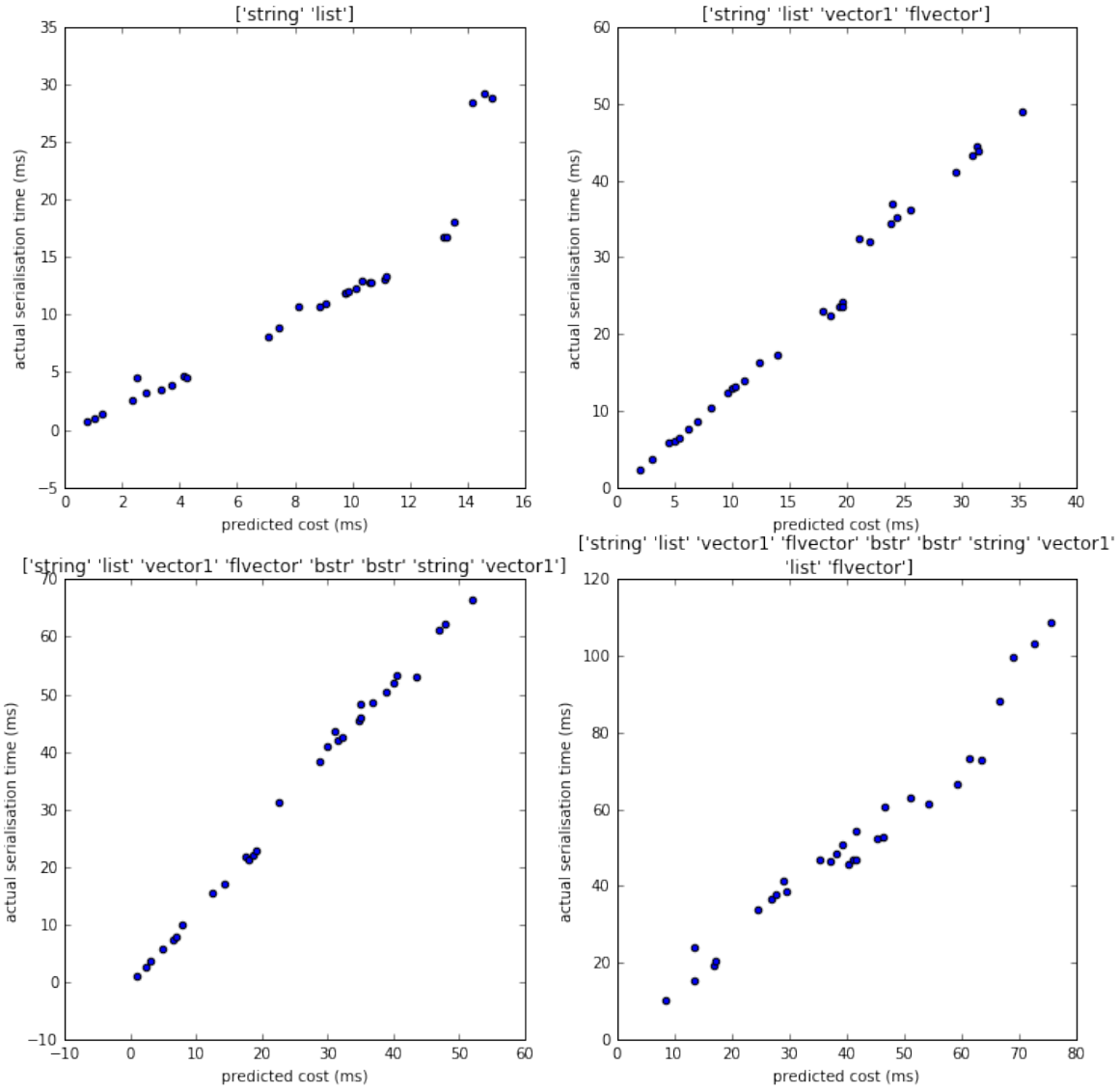

Figure 4.9: Actual serialisation time vs predicted serialisation time for heterogeneous tuples (Racket)

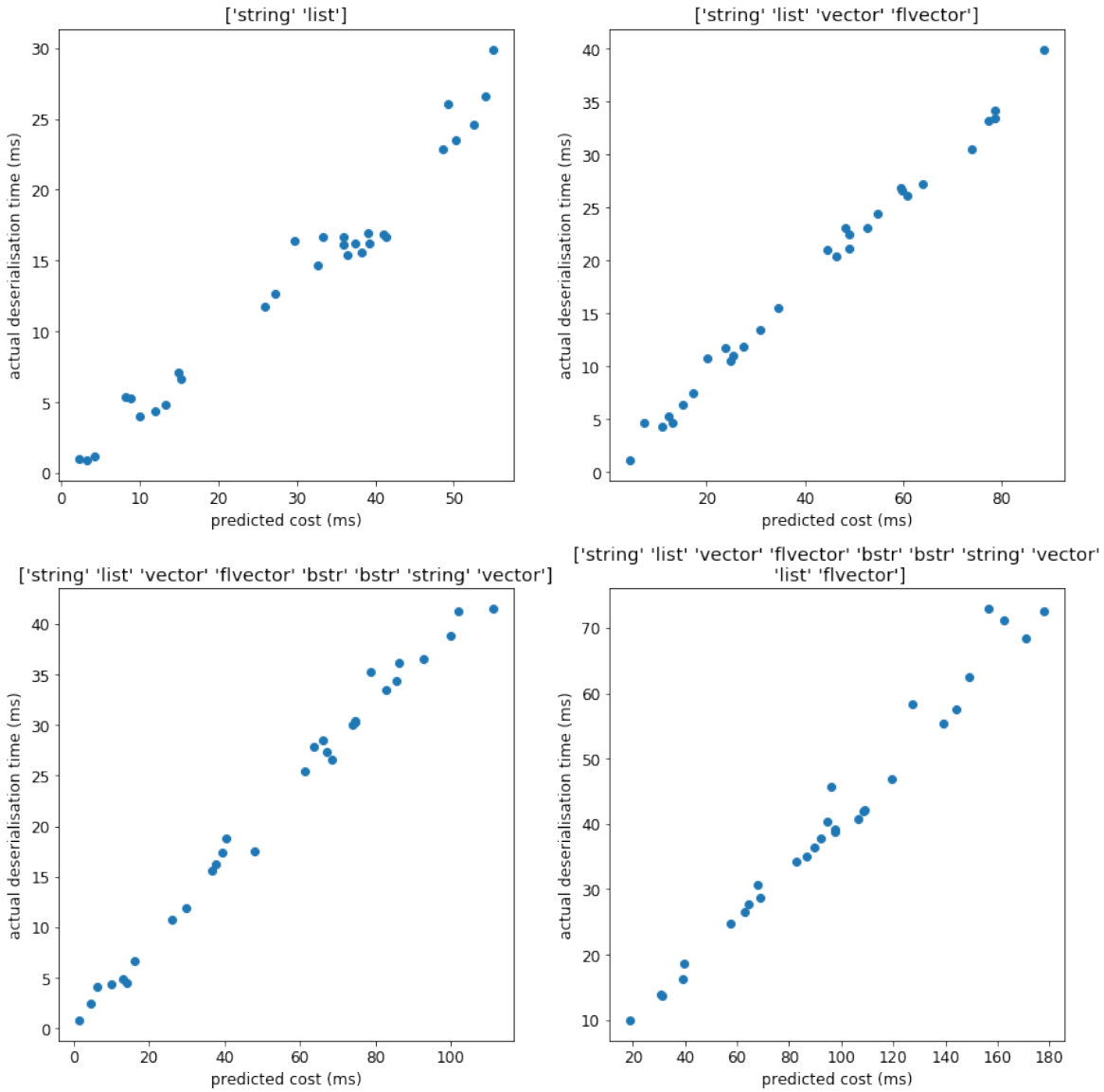

Figure 4.10: Actual deserialisation time vs predicted deserialisation time for heterogeneous tuples (Racket)

Type	Gradient (racket)	Intercept (racket)	Gradient (pycket)	Intercept (pycket)
'string' 'list'	1.4527	−2.8272	1.6359	−0.6187
'string' 'list' 'vector1' 'flvector'	1.0771	−2.4532	1.2957	0.0626
'string' 'list' 'vector1' 'flvector' 'bstr' 'bstr' 'string' 'vector1'	0.9001	−1.9354	1.1193	0.3294
'string' 'list' 'vector1' 'flvector' 'bstr' 'bstr' 'string' 'vector1' 'list' 'flvector'	1.0724	−6.0817	1.2278	0.2874

Table 4.8: Additive parameters

4.4 *ASL* Integration

The cost model is implemented to minimise any overhead from its calculation. The existing *ASL* serialisation and deserialisation code is modified so that it stores information about the type and size of the data which has been (de)serialised. (De)serialising composite types such as lists or vectors results in the types and sizes of all the constituents being stored also. The cost is calculated by using the type information to lookup the weightings in a hash table, and multiplying the weighting by the size of the data structure. For aggregate, the cost is calculated from recursive sum of the costs of the constituents.

After a completed send or receive of data, a `comms-send` or `comms-recv` event is generated, with the cost of the send or receive as its value. A function `total-comms-cost` retrieves all the `comms-send` and `comms-recv` events and sums them, giving the total comms cost up to that point in the execution of the *ASL* program.

Table 4.9: Validation Benchmarks

Benchmark
Primes
Sum Euler
Matrix Multiplication
Odd
Sequence Alignment

4.5 Cost Model Validation

This section describes the cross-validation of the parameterised cost model, implemented in *ASL* against real-world programs. It is hypothesised that the predicted communication cost will be directly proportional to the measured communications overhead and that this relationship will be a one-to-one correspondence. This will show that the model satisfies the requirement of costing arbitrary types (Section 4.1).

4.5.1 Benchmarks

The benchmarks used in the validation are shown in Table 4.9. These benchmarks were chosen to provide a range of different communication patterns i.e Sum Euler has very little data communication, while Matrix Multiplication sends and receives significant volumes of data.

4.5.2 Hardware and Software Environment

The benchmarks are run on GPGcluster, consisting of 16 2.0 GHz Xeon servers with 64 GB of RAM running Ubuntu 14.04. Each benchmark uses 4 nodes of the cluster. The FATA machine — a 2.6GHz Xeon server with 64GB of RAM — is also used to validate the model. Revision `d45e79919f` of branch `runtime_trace_analysis` of the *ASL* Pycket fork [98] are used.

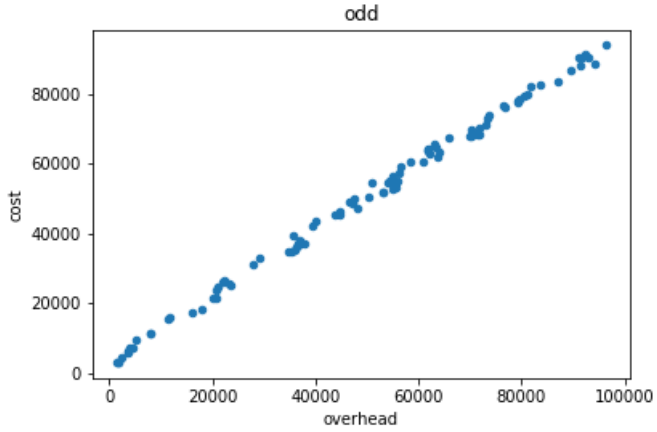

Figure 4.11: Plot of predicted communications costs vs actual overheads for odd filter — GPG platform

Table 4.10: Fit Gradients of Cross-validation Plots - GPG

Benchmark	Gradient
Odd	0.9402103590702535
Primes	97.23471323528817
matmul	4.676577511527445
euler	0.28252062453154586
seq	0.6026394666104261

4.5.3 Methodology

Each benchmark is run repeatedly with randomly generated inputs. The *ASL* runtime is modified to set the task granularity as small as possible - this minimises any adaptive behaviour which may cause variation in communications patterns. After each benchmark run, the predicted total communication cost and measured total overhead are recorded. The measured overhead is calculated by subtracting the execution time on each worker node from the total execution of the parallel program.

4.5.4 Results and Analysis

Figure 4.11 shows a clear linear relationship between the predicted communications cost and the measured communications overhead for the odd filter benchmark. The remaining figures can be found in Appendix B.3, all show the same linear relationship.

The gradients of the fits of each plot for each platform can be found in Tables 4.10 and 4.11. The gradients show some variation between benchmarks, but are all generally close together.

Table 4.11: Fit Gradients of Cross-validation Plots - FATA

Benchmark	Gradient
odd	0.4434298124236402
primes	0.4002519269231327
matmul	0.025140523138066674
euler	0.03846057406226092
seq	0.1215564168981652

Both GPG and FATA produce similar distributions of gradients, but the model on FATA seems to produce worse underestimations The variation could be the result of errors in the original training of the model or minor inaccuracies in the calculation of aggregate types. The data (de)serialised is generally in nested lists.

These results suggest that the communications cost model is an accurate predictor of communications overhead in the *ASL* system in real-world parallel programs.

4.5.5 Performance Overhead

The implementation of K in *ASL* is deeply intertwined with the serialisation and deserialisation algorithms. On Racket, K adds an overhead of 9.7% to (de)serialisation, with an overhead of 18.8% on Pycket. The performance overheads quoted above are strictly the extra time required to compute K compared to (de)serialisation alone. Integrating K does not affect performance of other parts of an *ASL* program.

4.6 Summary

The specifications outlined in Section 4.1, require a complex model. The final version of the cost model K, K_{tbsd}, satisfies these requirements. K_{tbsd} accounts for the use of both Racket and Pycket, including (de)serialisation and network transmission parameter values for both platforms. It accounts for the master-worker architecture defined in Section 2.7.1 by including parameter values for both serialisation and deserialisation. The additive property demonstrated in Section 4.3 and the validation in Section 4.5 show that K meets the need to cost arbitrary data types.

It is noted with interest that the calibration for K is dominated by (de)serialisation. It is believed that K constitutes the first runtime, dynamic cost model for serialisation and network communication.

Chapter 5

Combined Cost Modelling

Chapter 3 and Chapter 4 present computation (Γ) and communication (K) cost models, which have been shown to accurately model those respective program costs. This chapter presents a new model, T, that unifies Γ and K.

Since any parallel *ASL* program will have computation and communication components, it is necessary to combine the two cost models.

This chapter's hypothesis is that such a combined model can more accurately predict a good task size than the default *ASL* implementation or by using Γ alone. Note: the result is more general than for *ASL*. It may apply to other platforms.

This chapter presents a theoretical derivation of a unified cost model from the two previous cost models Γ and K, and theorises about how that unified model may be used to predict the optimal task granularity. Finally, the ability of the model to choose good task group sizes for real programs is demonstrated.

5.1 Deriving a Combined Cost Model

5.1.1 *ASL* Architecture

A full description of the *ASL* architecture is given in Section 2.7. The key aspects of the *ASL* architecture that must be accounted for by the combined model are reprised below. The basic architecture of *ASL* is a master/slave architecture. The master node runs on the

Racket virtual machine. The master node interprets the *ASL* program and builds a task graph from the skeleton structure. This task graph is an acyclic, directed bipartite graph, where the vertices are alternately *tasks* and *futures* and the edges are dependencies between them. A task graph is evaluated by applying the function in the task to its input future, until no unevaluated tasks remain. Tasks are generally not distributed individually to the worker nodes; connected sub-graphs of the task graph are grouped together into *task groups*. Task groups are distributed to available workers by the scheduler and the master node then waits for the result to be sent back to it. The worker node sends the result future back to the master along with statistical information, including the Γ cost of the tasks in the task group.

The AS scheduler attempts to produce task groups of a predetermined 'ideal' size. The scheduler feeds the returned Γ costs of tasks into a linear regression engine and uses the result to choose the optimal number of tasks for a task group.

5.1.2 Derivation of Combined Model

The cost models Γ (Section 3.4.2, Equation (3.4)) and K (Section 4.2.7, Equation (4.7)) must be integrated into a unified model for the *ASL* system while taking into account the architecture described in Section 5.1.1. This model is denoted with the symbol T.

This model is based on the following assumptions:

1. No task work is ever executed on the master node. If work was performed on the master node, it would be invisible to *ASL*'s costing. This assumption is always true, as the architecture of *ASL* schedules all work on the worker nodes.

2. The overhead caused by the JIT compiler for (de)serialisation is minimal. The (de)serialisation algorithms will be have to be warmed up on the Pycket worker nodes. However, since the same instance of Pycket is run on a single worker for the entire lifetime of the *ASL* program, this overhead will be amortised quickly.

3. Non-*ASL* network traffic is minimal. Other network traffic could interfere with *ASL* network transmission.

4. The network is never saturated. This has not been observed when developing the

network cost model K, but if it occurred, it could cause K to underpredict the transmission cost.

The development of the model starts with very simple abstractions and then adds complications. In the following equations, n and p refer to the number of groups and processors respectively. The symbol t denotes a single task, and g an average task group. ϕ represents a whole program.

In the simplest case there is a single sequential task t and only computation cost need be accounted for.

$$T(\phi) = \Gamma(t) \tag{5.1}$$

Equation (5.1) describes the model for a purely sequential program. Since there is only one task, there is only one task group, which constitutes the entire program i.e. $\phi = g = t$. With this model sequential *ASL* programs can be costed.

Equation (5.2) introduces parallelism, parameterised by n and p. In Equation (5.2), T is defined as the computational cost of an average task group in ϕ, $\Gamma(g)$, multiplied by the number of groups, n, divided by the number of processors, p. In the remaining equations, $\Gamma(g)$ is always the average cost.

$$T(\phi) = \frac{n}{p}\Gamma(g) \tag{5.2}$$

$$T_m(\phi) = nK_m(g) \tag{5.3}$$

$$T_w(\phi) = \frac{n}{p}(\Gamma(g) + K_w(g)) \tag{5.4}$$

Introducing communication into the model, there are now two components — the total cost on the master node (Equation (5.3)) and the total cost on the worker nodes (Equation (5.4)). In Equation (5.3), the total cost T on the master node is equal to the number of task groups n multiplied by the communications cost on the master for a task group $K_m(g)$ — based on the observation that serialisation and deserialisation of tasks on the master happens sequentially.

In Equation (5.4), the total cost T_w on the worker nodes is defined as the ratio of task groups n to processors p, multiplied by the sum of the computational cost of the task group $\Gamma(g)$ and the worker node component of the communications cost $K_w(g)$.

The total cost of the entire system is defined in Equation (5.5) as the maximum of the communication on the master node (Equation (5.3)) and the computation communication costs on the worker (Equation (5.4)).

$$T(\phi) = \max(T_m(p), T_w(p)) \tag{5.5}$$

The final, expanded version of the model is shown in Equation (5.6). Here, K_m and K_w are the communication costs for the master and worker, respectively. They are defined in terms of the equations in Section 4.2.

$$T(\phi) = \max(nK_m(g), \frac{n}{p}(\Gamma(g) + K_w(g))) \tag{5.6}$$

5.2 Determining Good Task Granularity

In this section, the combined model defined in Section 5.1 is used to determine a good task granularity for a given parallel program.

5.2.1 Definitions

O is defined as a function that returns the optimal task granularity for a given input cost function. The simplest example would be $O(t)$, where t is the parallel execution time. $O(t)$ returns the task group granularity which produces the minimal value of t over a range of task granularities. Other defined functions are $O(T)$ and its components $O(T_m)$ and $O(T_w)$; defined in terms of T, T_m and T_w (Equations (5.3), (5.4) and (5.6) respectively).

These definitions will be used throughout the remainder of this section.

Table 5.1: Benchmarks

Benchmark name	Short Name
Prime Filter	primes
Mandelbrot	mandel
Sum Euler	sumeuler
Batch Sequence Alignment	seq
Odd Filter	odd
Matrix Multiplication	matmul

5.2.2 Benchmarks

The benchmarks in this section (listed in Table 5.1) have been chosen to produce a range of communication and computation patterns. All these benchmarks are implemented as parallel programs using the *AS* library. *AS* reads an environment variable `$GRAN` that tells the scheduler to attempt to schedule groups that have an average execution time equal to the value of `$GRAN` in milliseconds.

Prime Filter uses the *par-filter* skeleton to apply a probabilistic primality test algorithm. In this case, a list of 1,000,000 candidate numbers, beginning at 100,000 are used as the input values. The *Miller-Rabin* primality test is used.

Mandelbrot is the classic Mandelbrot set problem. The benchmark consists of a square Mandelbrot set calculated in parallel, with the `par-map` skeleton applied to each row.

Sum Euler applies Euler's totient function over a list of integers and sums the results. This benchmark applies the totient function using `par-map` on each integer in the input list. The results are then summed using the `seq-reduce` skeleton.

Batch Sequence Alignment uses the *Smith-Waterman* algorithm to find the best alignments of a random input string against random test strings using the `par-map` skeleton.

Odd Filter is a simple benchmark originally developed to test the `par-filter` skeleton. It simply uses `par-filter` to return all the odd numbers from a list of input integers.

Matrix Multiplication uses `par-map` to multiply two matrices together. Parallelism is achieved by splitting one matrix into rows and applying `par-map` on each row.

5.2.3 Methodology

Each benchmark is executed for increasing values of `$GRAN`. The particular range of `$GRAN` values varies for each benchmark - values are chosen to cover the smallest possible task size for a given problem, and the largest practical task size, given the constraints of the computing resource used. The number of discrete values of `$GRAN` ranges from 138 to 1200. The execution time for the benchmark, the number of task groups, the number of tasks per group, Γ, K_w and K_m are all recorded for each benchmark run.

5.2.4 Platform

The benchmarks are run on GPG, consisting of 16 2.0 GHz Xeon servers with 64 GB of RAM and gigabit Ethernet running Ubuntu 14.04; and FATA, consiting of a 32-core 2.6Ghz Xeon with 64GB of RAM. Each benchmark uses 16 Pycket workers, one on each node of the cluster. Revision `d45e79919f` of branch `runtime_trace_analysis` of the Pycket fork [98] is used.

5.2.5 Results

A sample of the results are shown in Figures 5.1 through 5.7. Each figure shows results for a single benchmark for the indicated architecture, and shows a plot of execution time versus task granularity, and plots of T_m, T_w and T versus task granularity. The remaining graphs can be found in Appendix C.1.

5.2.6 Predicting Optimal Granularities

Looking at Figures 5.1 to 5.7, for nearly every benchmark there is a point where the total execution time is minimised with respect to task granularity. There is also a *phase change* point on the plots of T_m, T_w and T where the rate of decrease of each cost function declines rapidly, and the graph levels out, and these points appear to correspond with the minima in the plots of execution time against task granularity.

For each plot of execution time, there is generally an initial sharp fall in execution time with increasing task granularity, then a levelling off followed by a gradual increase in execution

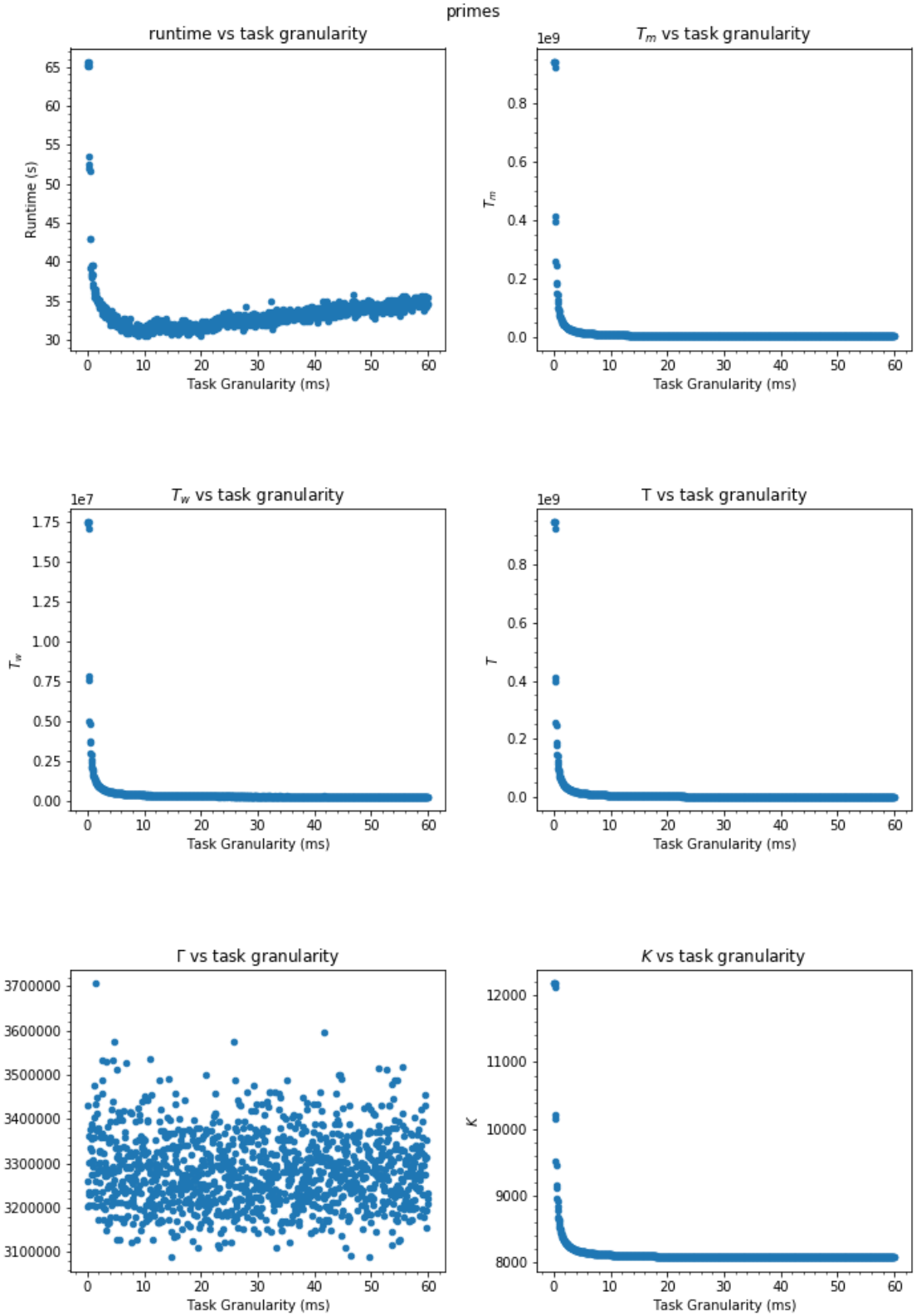

Figure 5.1: Prime Filter Results — GPG

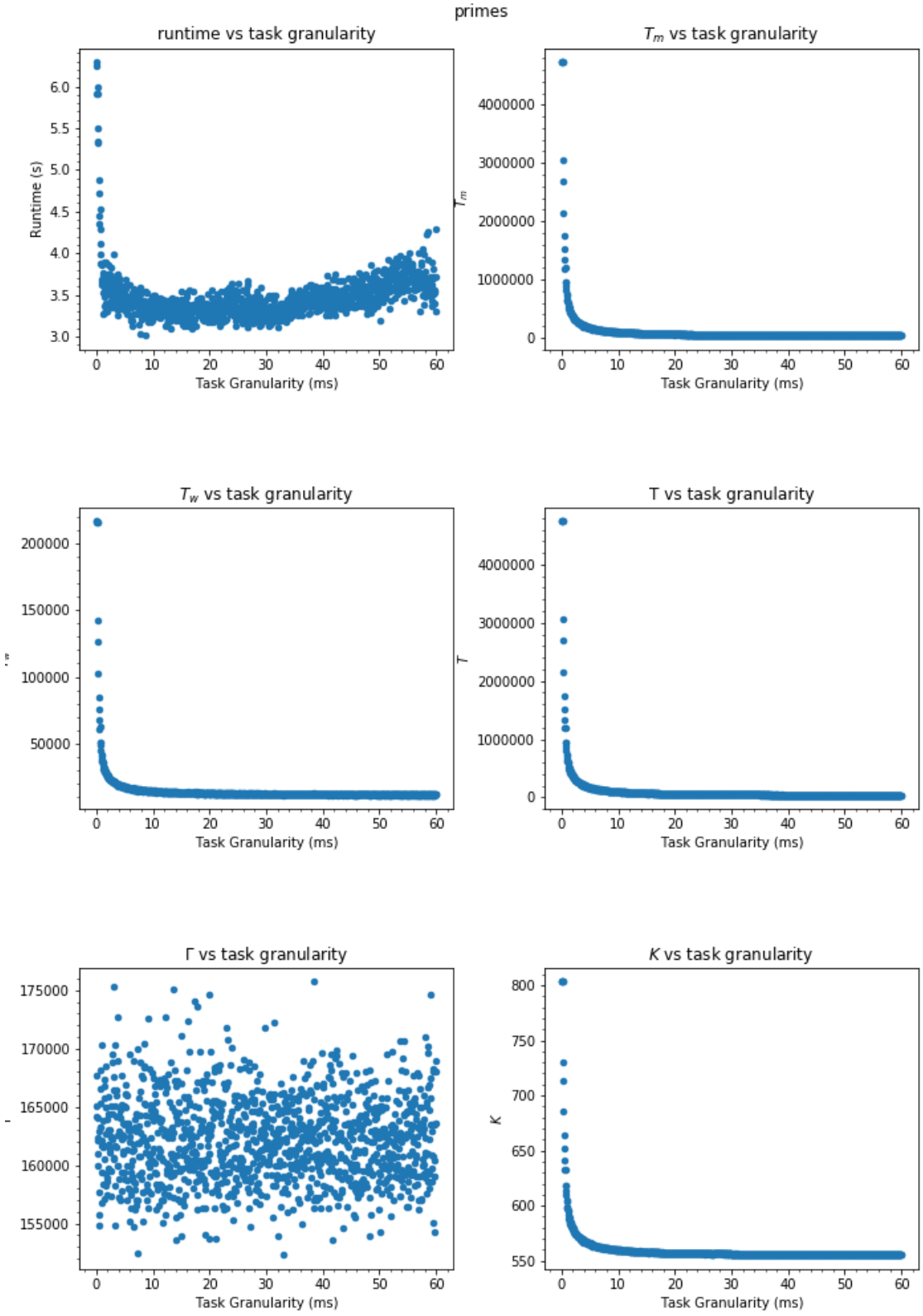

Figure 5.2: Prime Filter Results — FATA

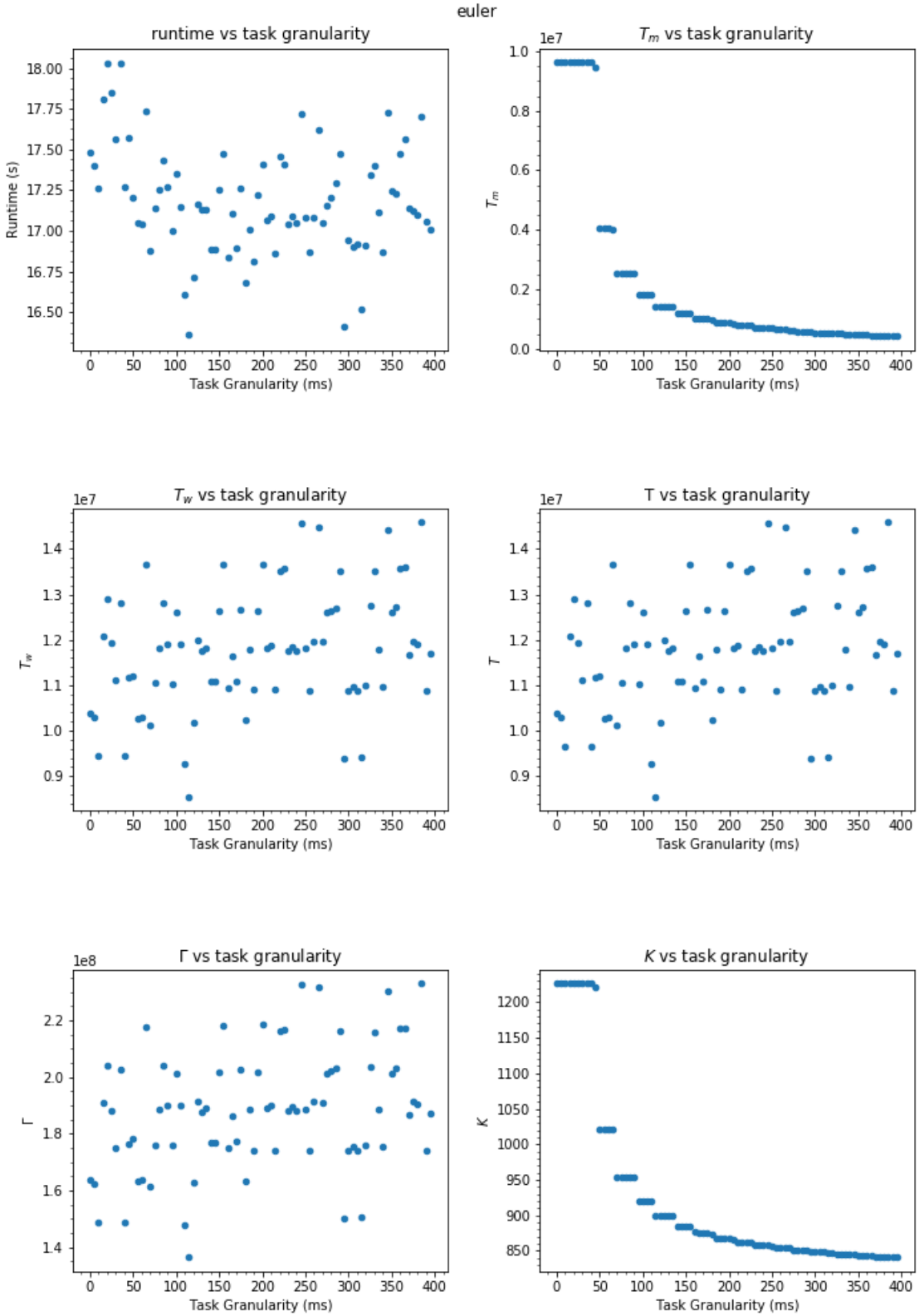

Figure 5.3: Sum Euler Results — GPG

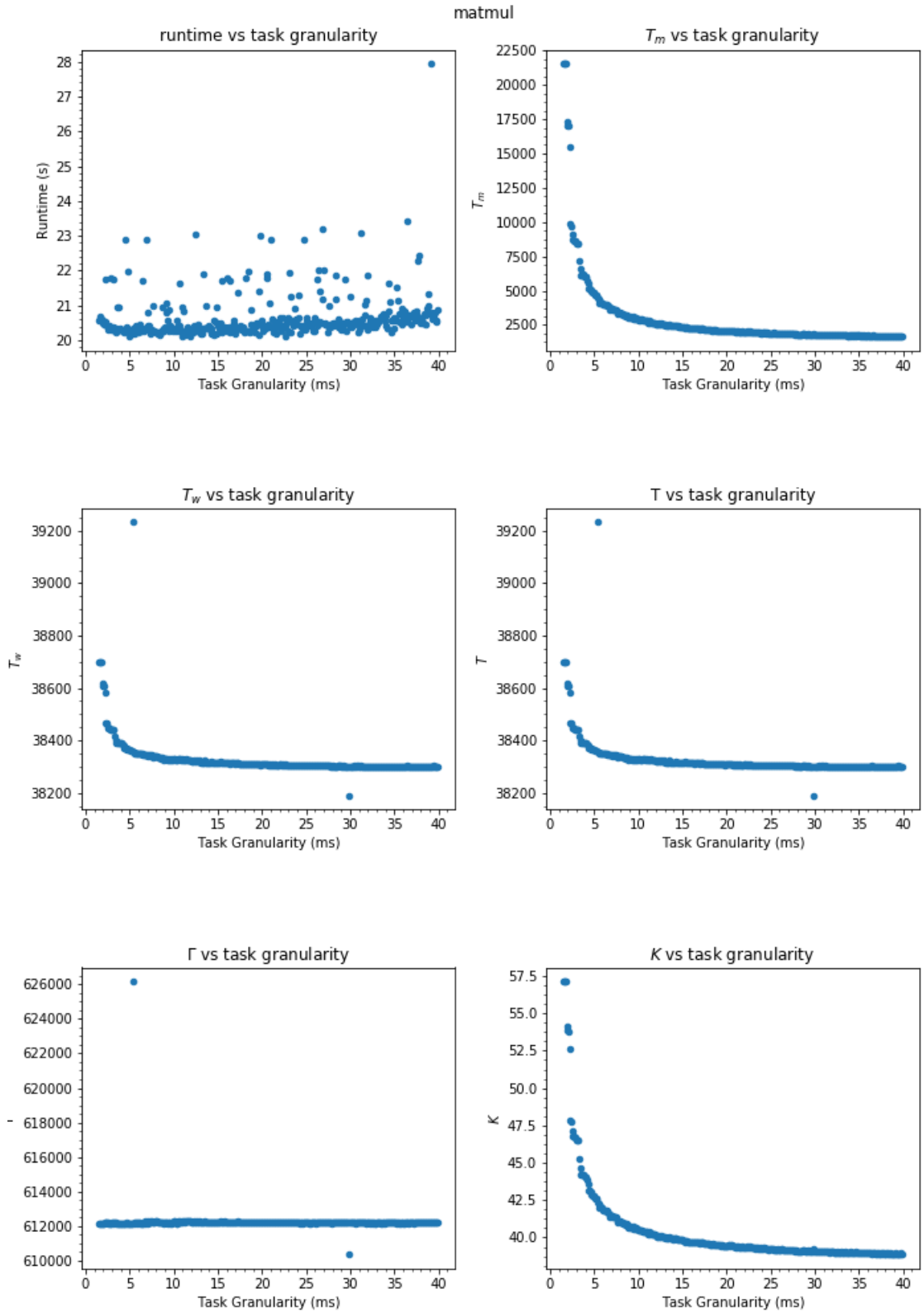

Figure 5.4: Matrix Multiplication Results — GPG

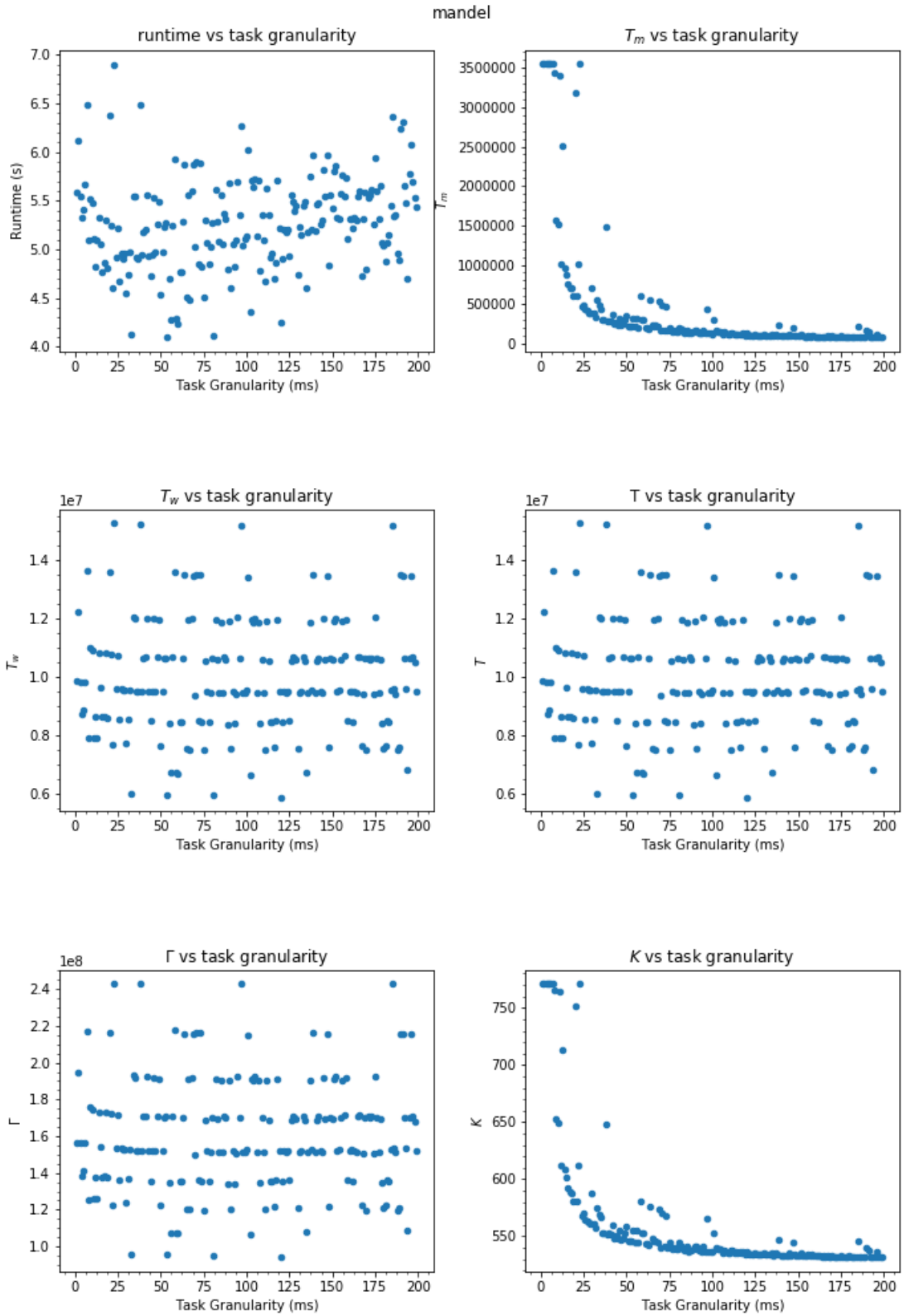

Figure 5.5: Mandelbrot Results — GPG

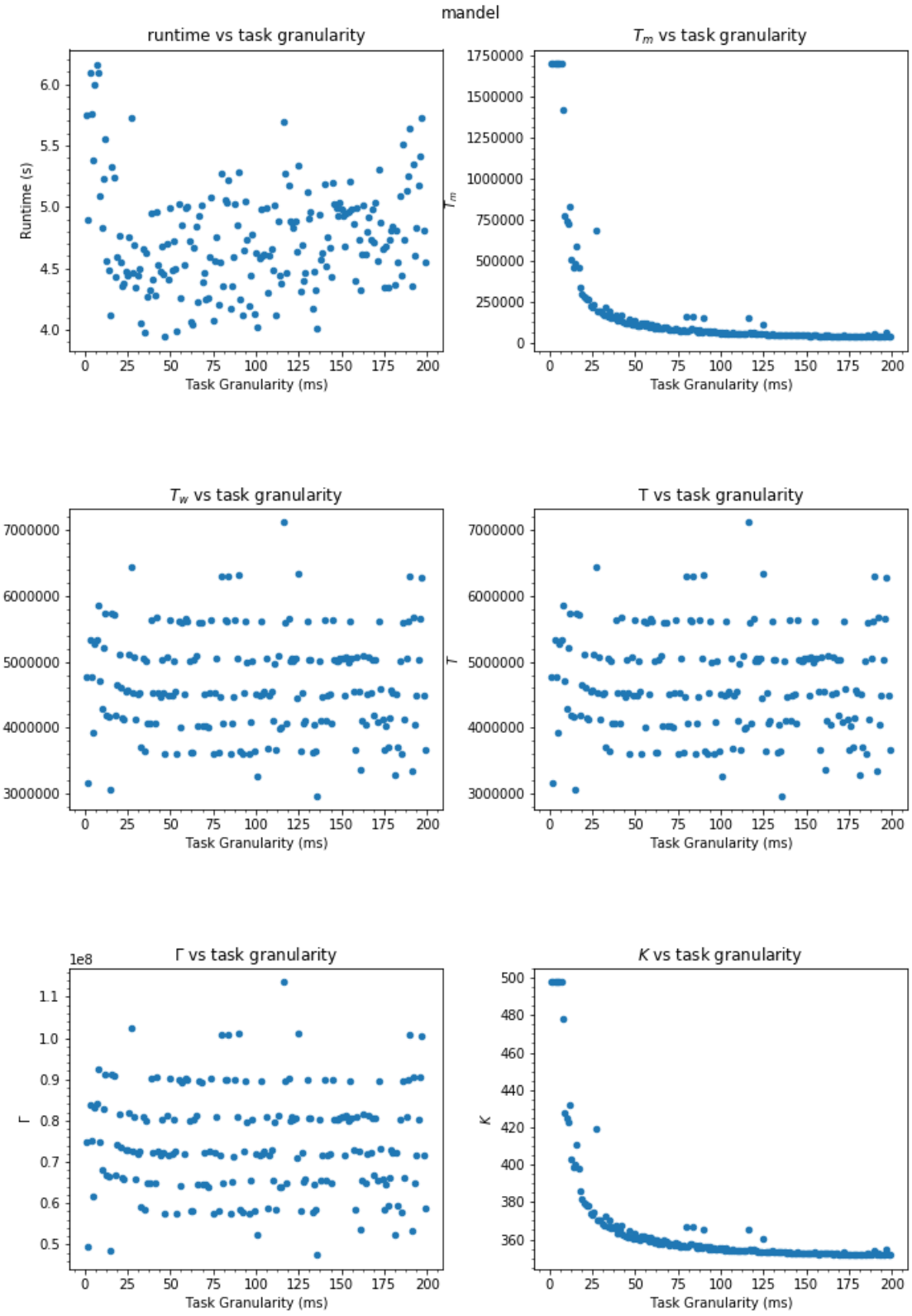

Figure 5.6: Mandelbrot Results — FATA

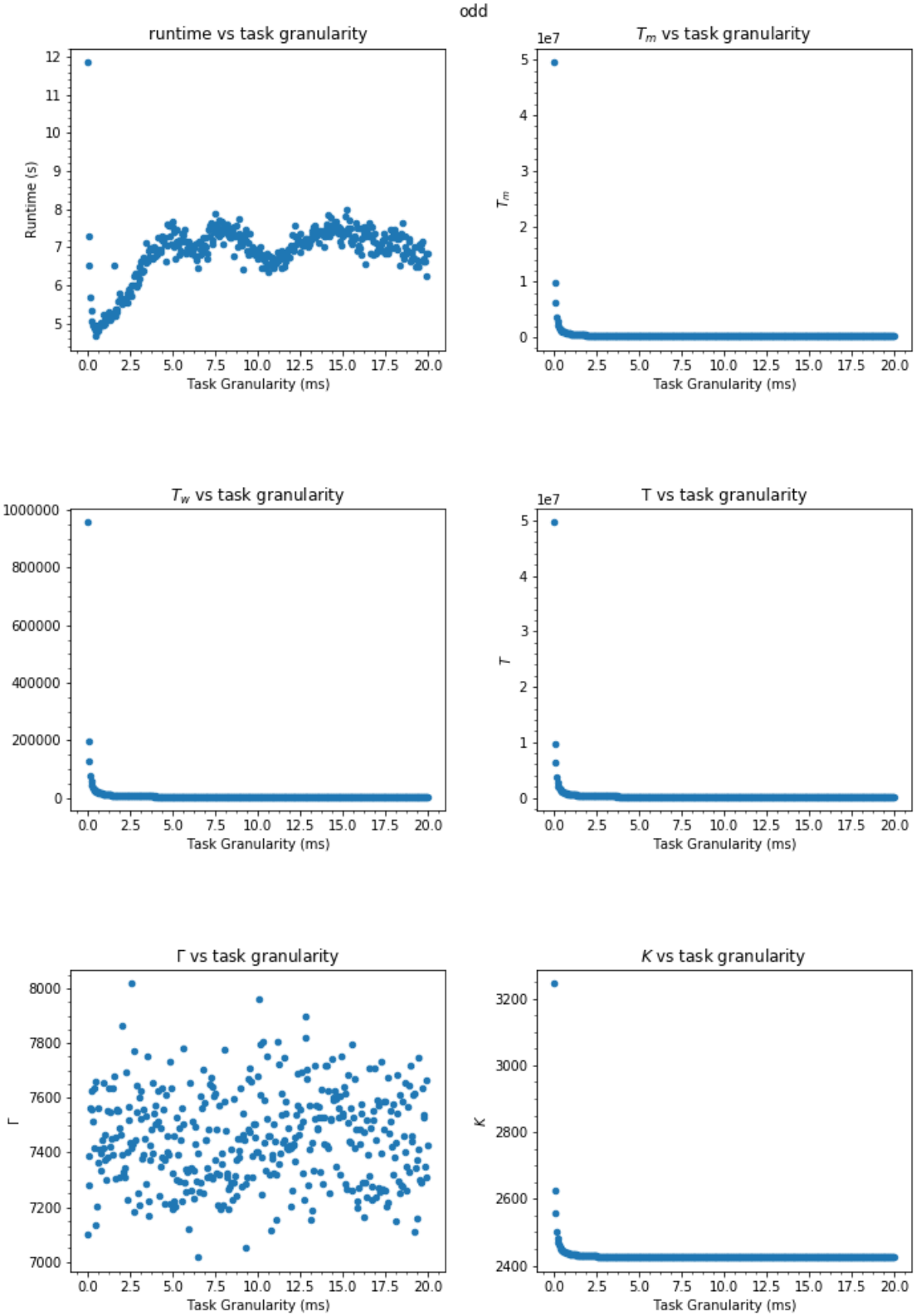

Figure 5.7: Odd Filter Results — GPG

time. It can then be deduced that the optimal task granularity ($O(t)$) is when the measured runtime is at a minimum.

Correspondingly, in the plots of T_m, T_w and T there is an initial sharp fall in the respective value of T_m, T_w or T with respect to task granularity, followed by a levelling off. However, there is no corresponding later increase in these plots. This is not of concern, as the first point where the gradient of the plot is flat corresponds with the $O(t)$ point from the plot of t vs granularity.

There is little observable difference between the results seen from the GPG and FATA platforms.

There are a few exceptions to these observed trends. Figure 5.5 shows a "banding" the plots of t, T_w and T. Since this banding is not visible in the plot of T_m, it is most likely the result of computation behaviour — the range of possible values for Γ is perhaps too high for a given task size to make a reasonable prediction (this is possibly to be excepted with calculating the Mandelbrot set). The range of execution times on this graph is very tight and the execution time values are noisy. It may be that changing granularity affects some programs less than others.

The SumEuler results in Figure 5.3 show a similar pattern to the Mandelbrot results. Again, the values for t and Γ are very noisy.

Figure 5.4 doesn't deviate much from the general case, except that its initial fall-off in t, T_m, T_w and T is much less dramatic.

Figure 5.7 shows an unusual graph for t, showing a second and third drop off after the first. Given the simplicity of this benchmark along with its constant communication and computation patterns, this behaviour is most likely a result of network interference, or I/O patterns.

The Mandelbrot, SumEuler and Odd filter observations are also seen in the FATA results.

Tables 5.2 and 5.3 contain values for $O(t)$, $O(T_m)$, $O(T_w)$ and $O(T)$ — the best task group granularity for the respective cost function — for each benchmark, along with the corresponding execution time at that predicted granularity, for the GPG and FATA platforms. The values for $O(t)$ are determined by reading the minimum values of t off the graph, while $O(T_m)$, $O(T_w)$ and $O(T)$ are determined by visual inspection of the graphs. The correspond-

Benchmark	$O(t)$	$O(T_m)$	$O(T_w)$	$O(T)$	$O(\Gamma)$	$O(K)$
primes	10.52	7	4	7	N/A	7
seq	9.61	8	5	8	36	8
sumeuler	N/A	150	N/A	N/A	N/A	150
matmul	N/A	10	5	5	N/A	10
mandel	54.0	50	N/A	N/A	N/A	50
odd	0.51	0.5	0.5	0.5	N/A	0.5

Table 5.2: Best Task Granularities — GPG

Benchmark	$O(t)$	$O(T_m)$	$O(T_w)$	$O(T)$	$O(\Gamma)$	$O(K)$
primes	8.67	4	4	4	N/A	4
seq	17.42	20	19	20	N/A	20
sumeuler	100	150	N/A	N/A	N/A	150
matmul	N/A	10	9	9	N/A	10
mandel	47.0	25	N/A	N/A	N/A	25
odd	0.61	0.6	0.6	0.6	0	0.6

Table 5.3: Best Task Granularities — FATA

		t at optimal granularity				*ASL*
Benchmark	t	T_m	T_w	T	Γ	K
primes	30.46	32.07	32.55	32.07	N/A	32.07
seq	2.72	3.04	3.26	3.04	3.34	3.04
sumeuler	16.36	17.04	N/A	N/A	N/A	17.04
matmul	20.11	20.53	20.54	20.54	N/A	20.53
mandel	N/A	5.07	N/A	N/A	N/A	5.07
odd	4.67	5.41	5.41	5.41	N/A	5.41

Table 5.4: Total execution times by using predicted granularity for each predictor cost model — GPG

		t at optimal granularity				*ASL*
Benchmark	t	T_m	T_w	T	Γ	K
primes	3.02	3.67	3.67	3.67	N/A	3.67
seq	2.83	2.93	2.96	2.93	3.34	2.93
sumeuler	15.52	16.01	N/A	N/A	N/A	16.01
matmul	15.01	15.38	15.39	15.39	N/A	15.38
mandel	3.95	4.83	N/A	N/A	N/A	4.82
odd	4.95	5.85	5.85	5.85	5.75	5.85

Table 5.5: Total execution times by using predicted granularity for each predictor cost model — FATA

Benchmark	t (T)	t (default)	speedup
primes	32.07	38.28	19.3%
seq	5.14	3.34	53.9%
sumeuler	N/A	N/A	N/A
matmul	20.54	26.18	27.4%
mandel	N/A	6.38	N/A
odd	5.41	6.36	17.6%

Table 5.6: Comparison of best times using T as a predictor with times from default *ASL* implementation — GPG

Benchmark	t (T)	t (default)	speedup
primes	3.67	34.83	849.04%
seq	2.93	4.73	61.4%
sumeuler	N/A	N/A	N/A
matmul	15.39	18.07	17.4%
mandel	N/A	5.87	N/A
odd	5.85	5.44	-7.5%

Table 5.7: Comparison of best times using T as a predictor with times from default *ASL* implementation — FATA

ing values for t at each predicted granularity are found in Tables 5.4 and 5.5. These values are calculated by interpolating the plot of t vs task group granularity data into a smoothing spline, and evaluating the spline function for the granularity read off the respective plot.

This table and the graphs in Section 5.2.5 show that the predictions of optimal task granularity from the combined model $O(T)$ and its components $O(T_m)$ and $O(T_w)$ produce a reasonable approximation of actual optimal task granularity read off from the graphs of t. It is interesting that in most cases, T_m is the best predictor of the best task group size, and also usually dominates the combined model T. This suggests that communications overhead is dominating these particular benchmarks.

Tables 5.6 and 5.7 compares the execution time at the predicted granularity using T with the actual execution time of the same program using the default configuration of *ASL* (the times for the default configuration are an average of 10 runs). The use of T produces speedups of 17.6 to 53.9% compared with the default version on GPG, and speedups from -7.5 to 849% on FATA[1]. For the benchmarks that could not be compared, using T_m as a predictor would have resulted in a similar speedup.

[1]The speedup for primes is so extreme that it must be the result of some experimental error.

It is interesting to note that generally, the predicted granularities on both FATA and GPG are similar, even with radically different parameter values for their cost models.

5.3 Summary

This chapter provides a motivation for and theoretical derivation of a combined computation and communication model (Section 5.1). The combined cost model T has been shown experimentally to make reasonable predictions of optimal task granularity, resulting in speedups of up to 54% on GPG (Section 5.2.5). T_m, however, seems to make more consistently reliable predictions, producing similar or identical task group granularities and speedups. These observations are seen on two different architectures. This suggests that master-side communication dominates, and it may be possible to ignore the computational cost entirely. However, the benchmarks used in this chapter are not exhaustive by any means, so the prudent course of action would be to use T to make predictions. The results in this chapter have shown a degree of performance portability can be achieved on different architectures.

Chapter 6

Conclusion

This chapter summarises and concludes this thesis. Section 6.1 summarises the research. Section 6.2 then outlines the limitations of the work, and Section 6.3 outlines potential directions for future work. Section 6.4 presents some concluding remarks.

6.1 Summary

Parallel programming is extremely challenging, and the problem of performance portability complicates it further. This work described in this thesis helps ease these problems by supporting the Adaptive Skeleton library (*ASL*) in adjusting task size implicitly, without any user intervention, other than selecting an architecture.

Achieving this required the development of the first JIT-based cost model, Γ, and a dynamic communications cost model Γ. These were combined into the unified cost model, T, that allows the prediction of good group sizes in *ASL*.

This research has been reported in *Resource Aware Computing* 2016 and *Functional High Performance Computing* 2016, and the primary research contributions are as follows.

The Development and Validation of the First Dynamic Computational Cost Model for JIT Traces Chapter 3 describes the design and implementation of a system for extracting JIT trace information from the Pycket JIT compiler. Three computational

cost models for JIT traces, ranging from very simple CM0 to the parametric CMw (Equations (3.1) to (3.3)) are defined. A regression analysis over 41 programs from the Pycket benchmark suite to automatically tune the architecture-specific cost model parameters for two architctures (Equations (3.7) and (3.8)). While Γ may not provide accurate absolute performance predictions (Figure 3.4), it is shown that the tuned cost model can be used to accurately predict the relative execution times of transformed programs using six benchmarks (Table 3.5). This model is integrated into the *ASL* system, and is applied to samples of *ASL* tasks after JIT warm-up.

The development and validation of communications cost models for AS Chapter 4 illustrates the development of the dynamic communications cost model, K, for the *ASL* system.

Equations (4.1) to (4.11) describe increasingly complex abstract cost models for modelling the communication cost, with the determined weightings shown in Tables 4.4 and 4.7.

Finally, it is shown that cost model instances for primitive data types can be combined to accurately predict cost models for compound data types (figs. 4.7 and 4.9 and table 4.8). Section 4.5 describes the cross validation of K on the benchmarks in table 4.9.

A unified cost model of jit computation and communication Chapter 5 develops a model, T (eq. (5.6)), that combines the cost models created in Chapter 3 and Chapter 4 — Γ and K.

Section 5.2.1 describes a theoretical method for predicting optimal group size using T and an experimental validation of this approach. The results in table 5.2 and fig. 5.1 show that T can be used to choose the correct *ASL* group size.

6.2 Limitations

The work described in this thesis is limited in a number of ways. There are also limitations of *ASL* itself that affect the work described here.

In Chapter 3, the computational cost model, Γ, does not make good absolute predictions of execution time between different benchmarks. While not strictly necessary, a cost model

which accurately made such predictions would be more useful outwith the context of this thesis or *ASL*. Related to this limitation, the regression model used to parameterise Γ has been tested only with the parameters described; it is possible that a simpler model, or one which grouped instructions in a different manner, could be more accurate.

Γ has been parameterised on two different hardware platforms. Testing on more platforms would allow stronger claims on performance portability.

Pycket was chosen as the language platform for this work and *ASL* early on in the life of the Adaptive Just-in-Time Parallelism (AJITPar) project, as it was one of the only reasonably mature trace-based JIT functional programming languages, and the original vision of AJITPar involved using this for equational rewriting transforms. Ultimately, these types of transforms were not used, so a more mainstream, mature tracing JIT language could have been chosen. This would have increased the usefulness of this work. Similarly, Pycket — although a relatively performant programming language — is not used in scientific computing, and similar languages, such as python, that are used in scientific computing, make heavy use of native code. If the work described here was to be seriously used in this domain, it would need to have some way of modelling calls to native code.

The results in Chapter 5 are based on a limited number of parallel benchmarks; adding extra benchmarks would add more credence to the results.

The network send/receive components of K were determined with the assumptions that there was minimal other activity on the network, and that the network never became congested. This limits the usefulness of the model in environments where there is other network activity or situations where the worker nodes send large results back to the master at exactly the same time.

Both models Γ and K used linear regression constrained through the origin. This was necessary as the unconstrained regression often resulted in negative y-intercepts. That type of result is nonsensical in this context, as it suggests an overhead which confers a speedup, or a negative time at an x of zero. However, this will inevitably affect the accuracy of the regression somewhat.

The determination of the optimal group sizes in Section 5.2.5 is carried out by manual inspection. This introduces the risk of both manual error and unconscious bias.

6.3 Future Work

The work detailed in this thesis addresses the problem of automatically determining the optimal group size in *ASL*, and also provides two novel cost models for computation and communication. This opens up possible avenues for future research, both in novel extensions to the work, and in addressing the limitations discussed in Section 6.2. This section outlines several of these potential future directions.

6.3.1 Support for other Programming Languages

As discussed in Section 6.2, the choice of Pycket as the language platform limits this thesis. Porting the work described here to a more mainstream platform, would create many more potential users. The obvious choice as a target language would be PyPy. First, this would allow much of the engineering described in this thesis to be reused. Second, PyPy is mostly compatible with Python, allowing a vast number of programs to potentially use the system. Third, porting *ASL* to a more mature platform such as PyPy would avoid the awkward implementation issue of having the master and worker nodes be entirely different language virtual machines. This would allow a number of improvements, including work stealing scheduling and warmup of tasks on the master nodes. Finally, porting to PyPy would allow access to the *SciPy* libraries [102], a popular library for scientific computing and data analysis in Python. This would require extending Γ to account for the native libraries included in SciPy. *ASciPY* was a proposed project to investigate applying the techniques of *ASL* to SciPy.

6.3.2 Improving Computational Cost Models

The computational cost model, Γ, makes accurate predictions of the effect on execution time of applying code transformations to a program. It is limited in how well it predicts execution time between programs. There is scope for potentially improving the absolute accuracy of Γ. First, training Γ on a larger set of benchmarks could improve accuracy. Second, reclassifying the JIT instructions, or creating additional more fine-grained instruction classes could increase accuracy, although this runs the risk of over-fitting to the training set.

The original model itself and the previously suggested improvements all assume linear independence between instructions i.e that the presence of one instruction does not affect the execution time of another. However, this is not necessarily always true; caching, pipeline effects or speculative execution could all result in interdependence between instructions. There are well known techniques for modelling these effects in code, and these could be applied to Γ.

6.3.3 Other Applications of Cost Models

The cost models developed for this thesis have potential to be useful in other contexts. Γ could be usefully be applied in the Pycket or PyPy compilers by tweaking the *hotness* threshold — the point at which the interpreter decides that a loop should be JIT compiled. As it stands, the PyPy/Pycket JIT compiles loops after a fixed number of iterations. Modifying this fixed number based on the cost of a loop measured using Γ could perhaps result in better performance. The same techniques could also be used on other platforms.

6.3.4 Other Approaches to JIT-based Parallelism

The work described in this thesis is one approach to applying JIT technology to parallelism, in this case using JIT traces to alter the degree of parallelism. It would be interesting to see an attempt to use trace-based JIT technology to safely identify possible parallelism or even automatically dynamically parallelise code. This could be achieved by analysing the access patterns in a JIT trace to identify dependencies between loop iterations, potentially allowing the loop to be split into parallel tasks.

6.3.5 Adjustments to Skeleton Code

The work described in Chapter 5 allows the optimisation of parallelism by adjusting the group size at the scheduler level. However, some programs may be hand-optimised by the user e.g using a parallel map with very large tasks, to minimise communication overhead. In such cases there is little that group size adjustment can achieve. There may be scope for

using the techniques in Chapter 5 to inform the user that their hand optimisation may not be optimal.

6.3.6 Cost Models for Unknown Hardware Platforms

Γ has only been parameterised for two hardware platforms, and the radical differences in those sets of parameters shows the effect moving from one hardware platform to another can have. It is not practical to train the models for every possible hardware platform or network environment; however, it may be possible to develop a toolkit that would allow a user to automatically train the cost models for their particular platform.

It would be interesting to train the cost model and evaluate the system on radically different hardware, such as a true supercomputer, or a cloud-based cluster made up of virtual servers. These would both have radically different communication and execution time cost environments, and hence Γ and K cost models. It would be particularly interesting to see how the network environment on a virtualised cluster could be modelled — this would require extensions to the cost models as a virtualised cluster will likely violate some of the assumptions in Section 5.1.2. It would also be interesting to apply this work to a low-power cluster, e.g. a Raspberry Pi cluster. In addition to work on the cost model, extra work would be needed to ensure that Pycket and the *ASL* system both work on an ARM processor.

6.3.7 Use in Production Environment

Although the work described in this thesis has been tested experimentally, the benchmarks used have all been small and simple. It would be of great value to apply *ASL* and the techniques in Chapter 5 to a production environment with long-running, complex parallel code. This would provide useful validation, as well as illuminating any unexpected issues.

6.4 Concluding Remarks

The work in this thesis addresses the challenges of parallel performance portability for JIT-compiled languages. It contributes lightweight computation (Γ) and communication (K)

cost models, and explores how they can be combined to improve task granularity in the ASL system.

The work shows that this dynamic, cost-model driven approach using JIT technology has promise, and that there are significant opportunities arising for future research.

Bibliography

[1] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[2] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[3] "AJITPar project," http://www.dcs.gla.ac.uk/~pmaier/AJITPar/, 2014, accessed: 2014-11-13.

[4] J. M. Morton, P. Maier, and P. Trinder, "Jit-based cost analysis for dynamic program transformations," *Electronic Notes in Theoretical Computer Science*, vol. 330, pp. 5–25, 2016.

[5] P. Maier, J. M. Morton, and P. Trinder, "Jit costing adaptive skeletons for performance portability," in *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. ACM, 2016, pp. 23–30.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[7] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 948–960, 1972.

[8] G. V. Wilson, *Practical Parallel Programming*, 1st ed., ser. Scientific and Engineering Computation Series. The MIT Press, 1995.

[9] A. Grama, A. Gupta, G. Larypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Pearson Addison Wesley, 2003.

[10] A. Barak and O. La'adan, "The mosix multicomputer operating system for high performance cluster computing," *Future Generation Computer Systems*, vol. 13, no. 4–5, pp. 361 – 372, 1998, hPCN '97. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X9700037X

[11] D. Ridge, D. Becker, P. Merkey, and T. Sterling, "Beowulf: harnessing the power of parallelism in a pile-of-pcs," in *Aerospace Conference, 1997. Proceedings., IEEE*, vol. 2. IEEE, 1997, pp. 79–91.

[12] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[13] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[14] M. Garland, S. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing in cuda," *IEEE micro*, vol. 28, no. 4, pp. 13–27, 2008.

[15] K. O. W. Group *et al.*, "The OpenCL specification," *version*, vol. 1, no. 29, p. 8, 2008.

[16] T. Maruyama and T. Hoshino, "A c to hdl compiler for pipeline processing on fpgas," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. IEEE, 2000, pp. 101–110.

[17] C. Iseli and E. Sanchez, "A c++ compiler for fpga custom execution units synthesis," in *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*. IEEE, 1995, pp. 173–179.

[18] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 73–83, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1945023.1945033

[19] R. Schooler, "Tile processors: Many-core for embedded and cloud computing," in *Workshop on High Performance Embedded Computing*, 2010.

[20] G. Chrysos, "Intel® xeon phi™ coprocessor-the architecture," *Intel Whitepaper*, vol. 176, 2014.

[21] C. Severance and K. Dowd, "High performance computing," 1998.

[22] "Sunway mpp," https://www.top500.org/site/50623, accessed 30th January, 2018.

[23] "Top500 list november 2017," https://www.top500.org/lists/2017/11/.

[24] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial intelligence*, vol. 8, no. 3, pp. 323–364, 1977.

[25] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[26] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross, "Exploiting task and data parallelism on a multicomputer," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93. New York, NY, USA: ACM, 1993, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/155332.155334

[27] "Java fork/join," https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, accessed 30th January, 2018.

[28] D. Clark, "Openmp: A parallel standard for the masses," *Concurrency, IEEE*, vol. 6, no. 1, pp. 10–12, 1998.

[29] K. Hammond, "Glasgow parallel haskell (gph)," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 768–779.

[30] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[31] T. White, *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.", 2009.

[32] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[33] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 123–169, 1998.

[34] H. Kasim, V. March, R. Zhang, and S. See, "Survey on parallel programming model," in *Network and Parallel Computing*. Springer, 2008, pp. 266–275.

[35] I. P. A. S. Committee *et al.*, "Ieee std 1003.1 c-1995, threads extensions," 1995.

[36] S. Oaks and H. Wong, *Java threads*. O'Reilly Media, Inc., 2004.

[37] A. Ho, S. Smith, and S. Hand, "On deadlock, livelock, and forward progress," *Technical Report, University of Cambridge, Computer Laboratory (May 2005)*, 2005.

[38] A. Tousimojarad and W. Vanderbauwhede, "Comparison of three popular parallel programming models on the intel xeon phi," in *European Conference on Parallel Processing*. Springer, 2014, pp. 314–325.

[39] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen, "Jamp: an implementation of openmp for a Java dsm," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2333–2352, 2007.

[40] W. L. George, J. G. Hagedorn, and J. E. Devaney, "Impi: making mpi interoperable," *Journal of Research of the National Institute of Standards and Technology*, vol. 105, no. 3, p. 343, 2000.

[41] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.

[42] T. El-Ghazawi and L. Smith, "Upc: unified parallel c," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 27.

[43] S. Marlow, R. Newton, and S. Peyton Jones, "A monad for deterministic parallelism," *ACM SIGPLAN Notices*, vol. 46, no. 12, pp. 71–82, 2011.

[44] P. W. Trinder, H.-W. Loidl, and R. F. Pointon, "Parallel and distributed Haskells," *Journal of Functional Programming*, vol. 12, no. 5, pp. 469–510, 2002.

[45] P. Maier, R. Stewart, and P. Trinder, "Reliable scalable symbolic computation: The design of SymGridPar2," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1674–1681. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480677

[46] S.-B. Schlz, "Single assignment c: efficient support for high-level array operations in a functional setting," *Journal of Functional Programming*, vol. 13, no. 6, pp. 1005–1059, 2003.

[47] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM Sigplan Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[48] "Timing analyis on code-level (TACLe)," http://www.tacle.eu, accessed: 2016-1-11.

[49] U. Dal Lago and R. Peña, *Foundational and Practical Aspects of Resource Analysis: Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*. Springer, 2014, vol. 8552.

[50] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers - Principles, Techniques, & Tools*, 2nd ed. Pearson Addison Wesley, 2007.

[51] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti *et al.*, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[52] J. Abella, D. Hardy, I. Puaut, E. Quinones, and F. J. Cazorla, "On the comparison of deterministic and probabilistic WCET estimation techniques," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE, 2014, pp. 266–275.

[53] V. Rodrigues, B. Akesson, M. Florido, S. M. de Sousa, J. P. Pedroso, and P. Vasconcelos, "Certifying execution time in multicores," *Science of Computer Programming*, vol. 111, pp. 505–534, 2015.

[54] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons, "Space profiling for parallel functional programs," *ACM Sigplan Notices*, vol. 43, no. 9, pp. 253–264, 2008.

[55] R. W. Kersten, B. E. Gastel, O. Shkaravska, M. Montenegro, and M. C. Eekelen, "ResAna: a resource analysis toolset for (real-time) JAVA," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 14, pp. 2432–2455, 2014.

[56] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 92–101.

[57] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," in *Building the Information Society*. Springer, 2004, pp. 377–383.

[58] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "Source-level execution time estimation of C programs," in *Proceedings of the ninth international symposium on Hardware/software codesign*. ACM, 2001, pp. 98–103.

[59] E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez, "Resource analysis: From sequential to concurrent and distributed programs," *FM 2015: Formal Methods*, pp. 3–17, 2015.

[60] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Cost analysis of object-oriented bytecode programs," *Theoretical Computer Science*, vol. 413, no. 1, pp. 142–159, 2012.

[61] M. Autili, P. Di Benedetto, and P. Inverardi, "A hybrid approach for resource-based comparison of adaptable Java applications," *Science of Computer Programming*, vol. 78, no. 8, pp. 987–1009, 2013.

[62] D. Aspinall, R. Atkey, K. MacKenzie, and D. Sannella, "Symbolic and analytic techniques for resource analysis of Java bytecode," in *Trustworthy Global Computing*. Springer, 2010, pp. 1–22.

[63] M. Hofmann, "Automatic amortized analysis," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2015, pp. 5–5.

[64] E. Albert, J. Correas, G. Puebla, and G. Román-Díez, "Incremental resource usage analysis," in *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*. ACM, 2012, pp. 25–34.

[65] G. Barthe, P. Crégut, B. Grégoire, T. Jensen, and D. Pichardie, "The MOBIUS proof carrying code infrastructure," in *Formal Methods for Components and Objects*. Springer, 2008, pp. 1–24.

[66] P. W. Trinder, M. I. Cole, K. Hammond, H.-W. Loidl, and G. J. Michaelson, "Resource analyses for parallel and distributed coordination," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 3, pp. 309–348, 2013.

[67] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proceedings of the tenth annual ACM symposium on Theory of computing*. ACM, 1978, pp. 114–118.

[68] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, "Logp: Towards a realistic model of parallel computation," in *ACM Sigplan Notices*, vol. 28, no. 7. ACM, 1993, pp. 1–12.

[69] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[70] D. B. Skillicorn and W. Cai, "A cost calculus for parallel functional programming," *Journal of Parallel and Distributed Computing*, vol. 28, no. 1, pp. 65–83, 1995.

[71] R. Rangaswami, "A cost analysis for a higher-order parallel programming model," 1996.

[72] E. Visser, "A survey of rewriting strategies in program transformation systems," *Electronic Notes in Theoretical Computer Science*, vol. 57, pp. 109–143, 2001.

[73] S. L. P. Jones, "Compiling Haskell by program transformation: A report from the trenches," in *Programming Languages and Systems - ESOP'96*. Springer, 1996, pp. 18–44.

[74] S. P. Jones, A. Tolmach, and T. Hoare, "Playing by the rules: rewriting as a practical optimisation technique in GHC," in *Haskell workshop*, vol. 1, 2001, pp. 203–233.

[75] N. Scaife, S. Horiguchi, G. Michaelson, and P. Bristow, "A parallel SML compiler based on algorithmic skeletons," *Journal of Functional Programming*, vol. 15, no. 04, pp. 615–650, 2005.

[76] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, and A. Elliott, "Cost-directed refactoring for parallel Erlang programs," *International Journal of Parallel Programming*, vol. 42, no. 4, pp. 564–582, 2014.

[77] I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown, and K. Hammond, "Discovering parallel pattern candidates in Erlang," in *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*. ACM, 2014, pp. 13–23.

[78] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[79] T. Schilling, "Trace-based just-in-time compilation for lazy functional programming languages," Ph.D. dissertation, University of Kent, April 2013.

[80] M. Pall, "The luajit project," 2008.

[81] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: PyPy's tracing JIT compiler," in *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 2009, pp. 18–25.

[82] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot™ client compiler for Java 6," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 1, p. 7, 2008.

[83] C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt, "Meta-tracing makes a fast Racket," 2014.

[84] B. J. Bradel and T. S. Abdelrahman, "The potential of trace-level parallelism in Java programs," in *Proceedings of the 5th international symposium on Principles and practice of programming in Java - PPPJ '07*. New York,

New York, USA: ACM Press, Sep. 2007, p. 167. [Online]. Available: http://dl.acm.org/citation.cfm?id=1294325.1294348

[85] Y. Sun and W. Zhang, "On-line Trace Based Automatic Parallelization of Java Programs on Multicore Platforms," *Journal of Computing Science and Engineering*, vol. 6, no. 2, pp. 105–118, Jun. 2012. [Online]. Available: http://koreascience.or.kr/journal/view.jsp?kj=E1EIKI&py=2012&vnc=v6n2&sp=105

[86] H. G. Rotithor, "Taxonomy of dynamic task scheduling schemes in distributed computing systems," *IEE Proceedings-Computers and Digital Techniques*, vol. 141, no. 1, pp. 1–10, 1994.

[87] J. Epstein, A. P. Black, and S. Peyton-Jones, "Towards Haskell in the cloud," in *ACM SIGPLAN Notices*, vol. 46, no. 12. ACM, 2011, pp. 118–129.

[88] K. A. Armih and M. K. Aswad, "A skeleton-based programming framework for heterogeneous parallel architecture," 2015.

[89] K. A. Armih, "Armih phd thesis," Ph.D. dissertation, 2015.

[90] I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown, and K. Hammond, "Discovering parallel pattern candidates in erlang," in *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*. ACM, 2014, pp. 13–23.

[91] I. Bozó, V. Fordós, D. Horpácsi, Z. Horváth, T. Kozsik, J. Köszegi, and M. Tóth, "Refactorings to enable parallelization." in *Trends in Functional Programming*. Springer, 2014, pp. 104–121.

[92] J. Enmyren and C. W. Kessler, "SkePU: a multi-backend skeleton programming library for multi-GPU systems," in *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM, 2010, pp. 5–14.

[93] C. Brown, H.-W. Loidl, and K. Hammond, "Paraforming: forming parallel haskell programs using novel refactoring techniques," in *International Symposium on Trends in Functional Programming*. Springer Berlin Heidelberg, 2011, pp. 82–97.

[94] R. McIlroy and J. Sventek, "Hera-JVM: a runtime system for heterogeneous multi-core architectures," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 205–222.

[95] "pycket-bench," https://github.com/krono/pycket-bench, 2015, accessed: 2015-03-31.

[96] T. Pape, "Benchmarking pycket against some Schemes," https://github.com/krono/pycket-bench, accessed: 2016-01-09.

[97] "Computer languages benchmark game," http://benchmarksgame.alioth.debian.org/, accessed: 2016-01-09.

[98] J. M. Morton, "Pycket fork," https://github.com/magnusmorton/pycket, accessed: 2016-01-09.

[99] P. Maier, J. M. Morton, and P. Trinder, "Towards an adaptive framework for performance portability," in *Pre-proceedings of IFL 2015*, Koblenz, Germany, 2015.

[100] ——, "Towards an adaptive skeleton framework for performance portability," December 2015, technical Report number TR-2016-001. [Online]. Available: http://eprints.gla.ac.uk/120235/

[101] J. M. Morton, "Trace analysis utilities," https://github.com/magnusmorton/trace-analysis, accessed 2016-01-09.

[102] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed 24th January 2018]. [Online]. Available: http://www.scipy.org/

[103] M. J. Crawley, *Statistics: an introduction using R*. West Sussex, England: John Wiley & Sons, 2014.

[104] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Upper Saddle River: Prentice Hall, 1995.

Appendix A

Cost Model Investigatory Work

Some of the work in Chapter 3 was driven by speculative experiments, or approaches which were superseded. This appendix details that work.

A.1 Pycket Benchmark Suite Analysis

We must understand the nature and classifications of the types of traces seen in the average Pycket program. To do so, we will analyse the JIT instructions found in the Pycket benchmark suite.

A.1.1 Whole Suite Analysis

A histogram of JIT operations, taken from traces generated by all the cross-implementation benchmarks and shown in Figure 3.3, shows that overall these traces are also dominated by "high-cost" instructions.

A.1.2 Program-level Analysis

Individual programs in the Pycket benchmarks suite show quite varying instruction distributions compared to that shown in Figure 3.3, though they are still dominated by guards and object operations. Using k-means analysis, these programs can be divided into two clusters:

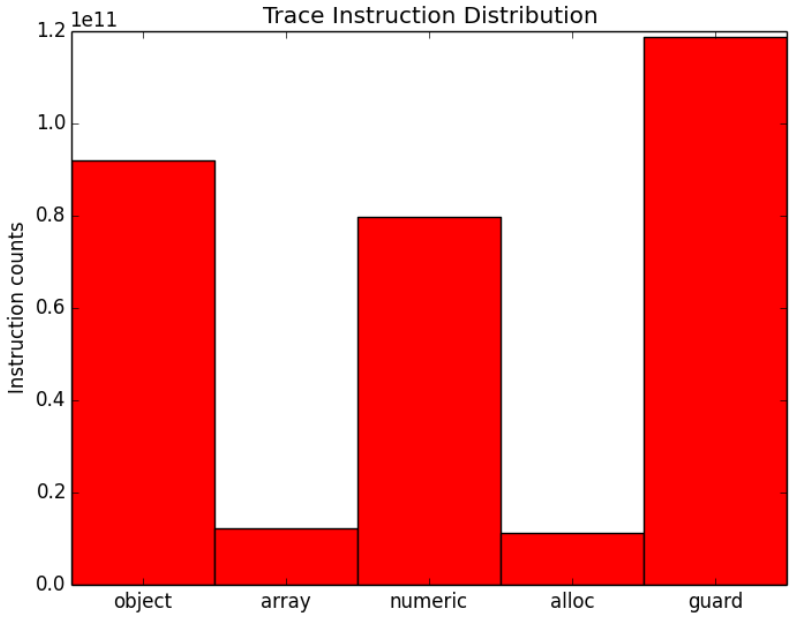

Figure A.1: Most common instructions in cross-implementation Pycket benchmarks

Cluster 1	Cluster 2
ack, array1,fib, fibc, pnpoly, sum, sumloop, trav2, fibfp, sumfp,	boyer, cpstak, ctak, dderiv, deriv, destruc, diviter, divrec, lattice, nboyer, perm9, primes, puzzle, sboyer, tak, takl

Table A.1: Clusters for whole benchmarks

numeric and non-numeric. The numeric programs still have a significant proportion of object operations. In Table A.1 Cluster 1 contains nearly all numerical benchmarks.

A.1.3 Trace-level Analysis

Looking at the 32013 individual trace fragments in the Pycket benchmark suite, a lot more variation is seen compared to the variation between the whole program histograms. *k-means* clustering shows 3 distinct clusters, the centroids of which are shown in Table A.2.

Traces in cluster 1 outnumber both 2 and 3 combined and are again dominated by object instructions and guards. From the centroid of cluster 2 we can see that the proportion of allocation instructions is much higher; this could correspond to the “cleaning up” portion

Cluster	count	object (%)	array (%)	numeric (%)	alloc (%)	guards (&)	jumps (%)
1	17946	38	0.15	0.69	5.6	51	4.0
2	9934	59	0.12	0.51	19	16	4.8
3	4133	22	4.8	11	1.6	54	6.8

Table A.2: Trace fragment centroids

of a trace where previously unboxed primitives are boxed again or possibly `cons` functions calls. Cluster 3 contains a higher proportion of array and numerical instructions.

A.2 Cost Model Search

This section describes previous attempts at attempting to determining the weights for the cost model CM_W. These approaches were necessary given the performance restrictions of checking each proposed model at the time.

To use the abstract weighted cost model CM_W (section 3.4.1), it is necessary to find values for each of the five weight parameters in equation 3.3. Rather than simply guess at appropriate values, we can systematically search the parameter space for an optimal solution. To do this a set of benchmarks are required, along with a search approach and a means of checking the accuracy of the cost model.

A.2.1 Performance Benchmarks

Using the cross-implementation benchmark suite from pycket-bench (section A.1), with the addition of the Racket *Programming Languages Benchmark Game*[97] benchmarks, the execution times and trace logs for each benchmark are recorded. The execution times are the average of 10 runs.

The platform is an Ubuntu 15.04 system with an Intel Core i5-3570 quad-core 3.40 GHz processor and 16GB of RAM. The Pycket version is revision 5d97bc3f of the `trace-analysis` branch of our custom fork[98], built with Racket version 6.1 and revision 72b01aec157 of PyPy. A slightly modified version of pycket-bench is used.

A.2.2 Model Accuracy

By applying a instance of a cost model to the trace output from the benchmark runs, the execution time for each benchmark can be plotted against the cost for that benchmark calculated using equation 3.4.The accuracy of the cost model is calculated by applying linear regression to the plot to obtain a linear best fit. The value of r^2, or the coefficient of determination[103], is used as an estimate of model accuracy; the higher the value the better the fit, and therefore the more accurate the cost model. The linear regression calculation is implemented using the SciPy library to enable automation.

A.2.3 Exhaustive Search

An exhaustive search of a part of the weight parameter space can be carried out by systematically varying the weights in equation 3.3. Representing the weights as a vector $\langle a, b, c, d, e\rangle$, the search covers all integral vectors between $\langle 0, 0, 0, 0, 0\rangle$ and $\langle 10, 10, 10, 10, 10\rangle$. On termination, the search returns the weight vector for the most accurate cost model (i.e. the model with the highest r^2 coefficient) in the given parameter space.

A.2.4 Genetic Algorithm Search

Unfortunately, the search space of the exhaustive search grows very quickly with the size of the bounds on the weight parameter space. While a bound of 10 is still feasible, exhaustively searching a paramter space to a bound of 100 is no longer possible. Fortunately metaheuristic search methods allow large search spaces to be covered relatively quickly.

Genetic Algorithms(GA) [104] are a set of meta-heuristics applied to search problems which attempt to mimic natural selection. Rather than exhaustively search the problem space, genetic algorithms attempt to evolve an optimal solution from an initial population. Genetic algorithms use a *fitness function* to evaluate the quality of a solution. The search process consists of a number of generations, in which the entire population is evaluated according to the fitness function and the fittest surviving to the next generation or being selected to reproduce and generate children for the next generation. Reproduction involves selecting any number of solutions from the population and combining them to produce a new solution

which contains aspects of its "parents". Random mutation is added to increase the diversity of the population. The search can be run for a fixed number of generations, until a sufficiently optimal solution is found, or until the population converges.

Genetic Algorithms are chosen as our metaheuristic as the coefficient of determiniation r^2 of linear regression is a useful fitness function, and the vector components map well to the idea of a "chromosome". Details of the search procedure are as follows.

- The *population* is a set of 40 weight parameter vectors $\langle a, b, c, d, e \rangle$. The first generation is completely random; subsequent generations are produced by selection, crossover and mutation, as described below.
- The *fitness function* is simply the r^2 value from the linear regression of the benchmark execution times against the benchmark costs (according to the cost model beign evaluated).
- Each new generation contains the fittest vector from the previous generation. Other vectors in each generation are created by
    1. *selecting* two parent vectors from the previous generation by "tournament selection" (where the fittest of two randomly chosen vectors survives to become a parent),
    2. producing a child vector by *crossing over* the parent vectors component-wise at random, and
    3. randomly *mutating* components of the child vector at a rate of 10%.
- The search terminates at 30,000 generations, returning the weight vector for the most accurate model found so far.

Subsampling

Many of the benchmarks in the benchmark suite are intended to test specific Scheme language features or JIT performance, and some benchmarks perform markedly different from the majority when analysed with the null and counting cost models CM_0 and CM_C. This raises the possibility that the benchmark suite contains outliers that will weaken the linear

regression of any possible weighted cost model. We use random sub-sampling whereby 8 randomly selected benchmarks are removed from each search, in order to account for the possibility of outliers in the benchmark suite. The best cost model reported is the best model found for *any* of 125 tested benchmark samples.

A.2.5 Search Results

Exhaustive Search

The cost model found by exhaustive search is displayed in equation A.1.

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in array \cup guard \cup debug \cup object \\ 1, & \text{if } x_i \in numeric \\ 10, & \text{if } x_i \in alloc \end{cases} \tag{A.1}$$

Genetic Algorithm Search

The cost model found by Genetic Algorithm search and subsampling is described in equation A.2.

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in debug \\ 34, & \text{if } x_i \in array \\ 590, & \text{if } x_i \in numeric \\ 9937, & \text{if } x_i \in alloc \\ 14, & \text{if } x_i \in guard \\ 211, & \text{if } x_i \in object \end{cases} \tag{A.2}$$

$$\gamma = \sum_{i=0}^{n} \begin{cases} 0, & \text{if } x_i \in debug \\ 2.43, & \text{if } x_i \in array \\ 42.1, & \text{if } x_i \in numeric \\ 709.8, & \text{if } x_i \in alloc \\ 1.00, & \text{if } x_i \in guard \\ 15.1, & \text{if } x_i \in object \end{cases} \tag{A.3}$$

The benchmark sample excluded the benchmarks ack, divrec, fib, fibfp, heapsort, lattice, tak, and trav2.

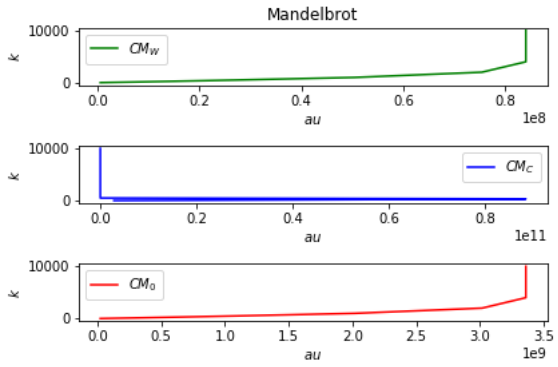

Figure A.2: k vs τ for Mandelbrot benchmark — FATA

The normalised version of this cost model, where the smallest non-zero weight is one, is shown in equation A.3. This is similar to the cost model found with using exhaustive search; the ratio between the allocation and numeric weighting is 16.84 in equation A.2 and 10 in equation A.1.

The cost model in Equation A.3 suggests that allocation instructions are the greatest contributor to program execution time followed by numeric instructions. The relatively high weighting of the numerical instructions in this model is interesting, as numerical computation is expected to take significantly less time than the reads and writes seen in object operations; however, the fact that numeric types are required to be boxed and unboxed, resulting in allocations and object reads and writes could account for this weighting.

The results described here are broadly in line with those detailed in Section 3.4.

A.3 Costing Transformations

This section contains graphs corresponding to those in Section 3.5.1, but for the FATA machine. Figures A.2 to A.4 represent the Mandelbrot, SumEuler and k-means benchmarks, respectively.

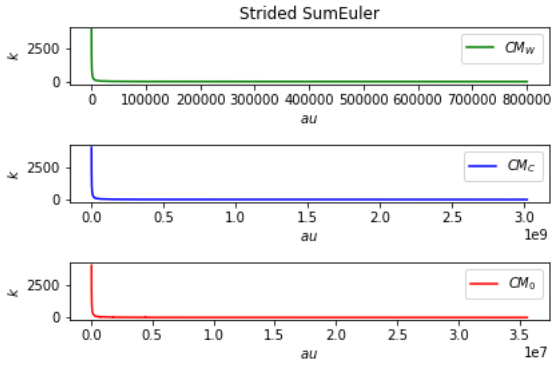

Figure A.3: k vs τ for SumEuler benchmark — FATA

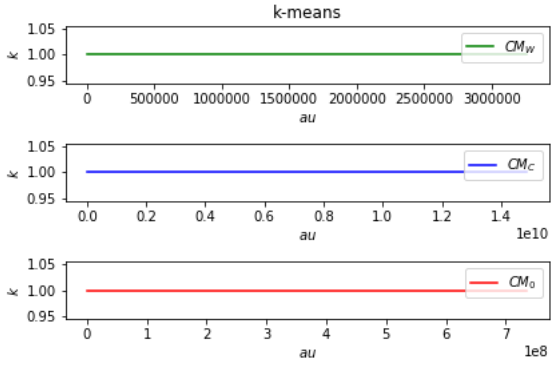

Figure A.4: k vs τ for k-means benchmark — FATA

Appendix B

Communications Modelling

This appendix contains details of superseded early experiments for the development of the communications cost model K in Chapter 4.

B.1 Constant Overhead Model

The original final definition for K, K_{tbsd} is shown in Equations (B.1) and (B.7). This definition originally had a constant overhead term in the equation.

The original parameter values for this equation (the gradients of fits for (de)serialisation and network graphs) are found in Tables B.1 and B.3

Unfortunately, many of the values for the intercept (the constant overhead term) were nonsensical — some were many orders of magnitude greater than the parameter value; others were negative.

The solution is to constrain the linear regression through the origin, producing the results shown in Chapter 4. This makes conceptual sense: a communications cost model should predict a cost of zero when serialising or sending nothing.

The decision to constrain regression through the origin is justified by the results of the validation in Section 4.5.

$$K_{tbsd}(x :: t) = K_r(x) + K_p(x) \tag{B.1}$$

Worker	Gradient (ms/byte)	Intercept (ms)
Racket	9.6907×10^{-7}	-6.1533×10^{-4}
Pycket	6.0045×10^{-7}	1.4075×10^{-2}

Table B.1: Network Send Gradients

Type	Gradient (racket) (ms/byte)	Intercept (racket) (ms)	Gradient (pycket) (ms/byte)	Intercept (pycket) (ms)
bstr	1.2367×10^{-6}	0.0064	1.7615×10^{-6}	0.0142
flmatrix	1.8157×10^{-5}	0.2172	6.4511×10^{-6}	0.5431
float	N/A	0.0164	0.0009	0.0083
flvector	1.8667×10^{-5}	0.2047	6.3731×10^{-6}	0.1607
int	N/A	N/A	0.0006	N/A
list	2.1906×10^{-5}	0.1852	8.0571×10^{-6}	0.1846
string	7.2733×10^{-5}	-0.0588	4.4195×10^{-6}	0.1145
vecbytes	1.1743×10^{-6}	0.3441	2.0331×10^{-6}	0.2974
vecstring	4.7208×10^{-5}	0.3643	4.5363×10^{-6}	0.3712
vector1	2.5946×10^{-5}	0.1930	9.0523×10^{-6}	0.2250
vector2	2.5632×10^{-5}	0.4515	6.0556×10^{-6}	0.4790

Table B.2: Serialisation parameters

Type	Gradient (racket) (ms/byte)	Intercept (racket) (ms)	Gradient (pycket) (ms/byte)	Intercept (pycket) (ms)
bstr	2.5506×10^{-5}	-0.0626	1.1498×10^{-7}	0.0086
flmatrix	1.1872×10^{-5}	-0.1852	4.0411×10^{-6}	0.3627
float	N/A	N/A	N/A	N/
flvector	1.0047×10^{-5}	0.0430	3.6638×10^{-6}	0.0813
int	N/A	0.0011	N/A	N/A
list	1.5609×10^{-5}	0.0237	1.7283×10^{-5}	0.0044
string	1.7734×10^{-6}	0.1403	1.3424×10^{-6}	0.0083
vecbytes	1.1661×10^{-6}	0.0283	3.4553×10^{-7}	0.1505
vecstring	7.4416×10^{-6}	-0.1677	1.4151×10^{-6}	0.1223
vector1	1.2883×10^{-5}	0.0350	1.0599×10^{-5}	0.1363
vector2	1.4444×10^{-5}	-0.0308	1.0355×10^{-5}	0.2020

Table B.3: Deserialisation parameters

$$K_r(x :: t) = l(x)(n_r + sd_r(t)) + o_{n,r} + o_{sd,r}(t) \tag{B.2}$$

$$K_p(x :: t) = l(x)(n_r + sd_p(t)) + o_{n,p} + o_{sd,p}(t) \tag{B.3}$$

$$sd_r(t) = s_r(t) + d_p(t) \tag{B.4}$$

$$sd_p(t) = s_p(t) + d_r(t) \tag{B.5}$$

$$o_{sd,r}(t) = os, r(t) + od, p(t) \tag{B.6}$$

$$o_{sd,p}(t) = os, p(t) + od, r(t) \tag{B.7}$$

B.2 FATA for Development of Communication Cost Model

This section includes the corresponding FATA graphs for Section 4.2

B.3 Communications Cost Model Validation

This section contains the remaining graphs for the validation of the cost model K for Section 4.5.

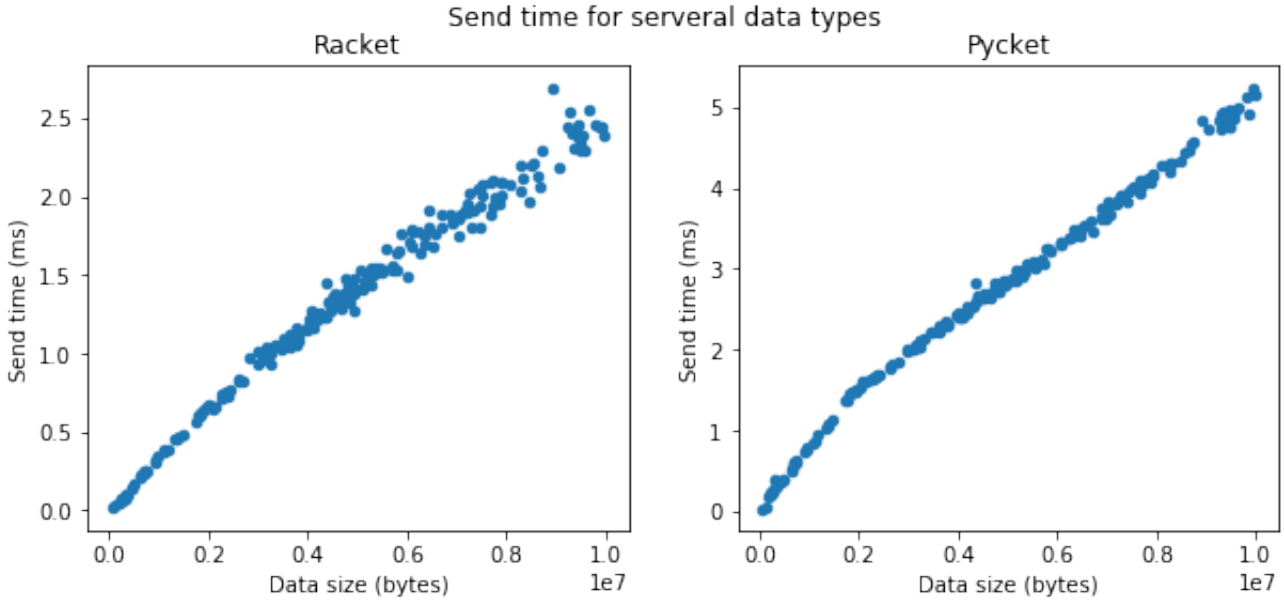

Figure B.1: Network send time results, FATA, loopback

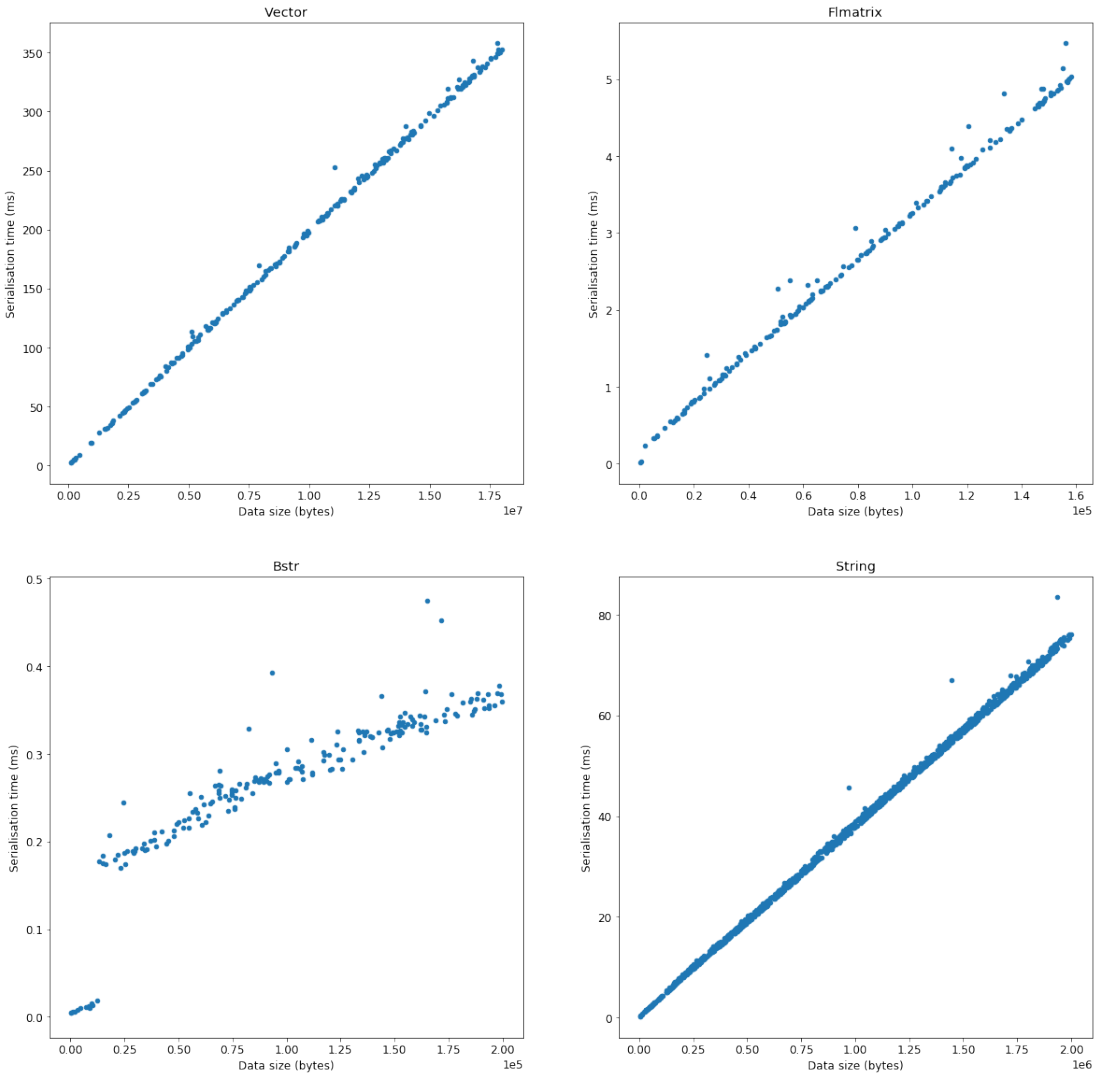

Figure B.2: Serialisation Time against Data Size separated by type (Racket; FATA)

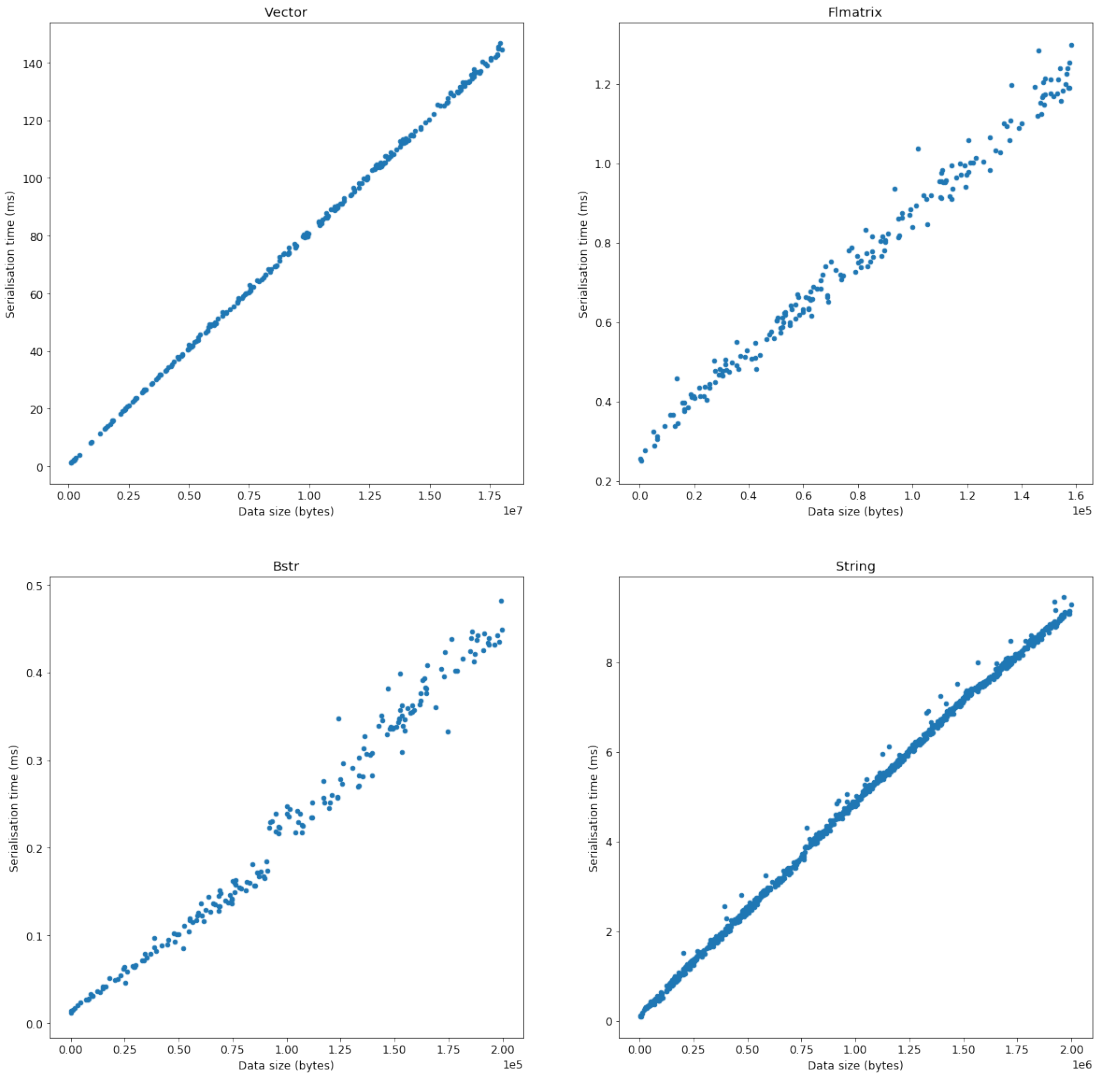

Figure B.3: Serialisation Time against Data Size separated by type (Pycket; FATA)

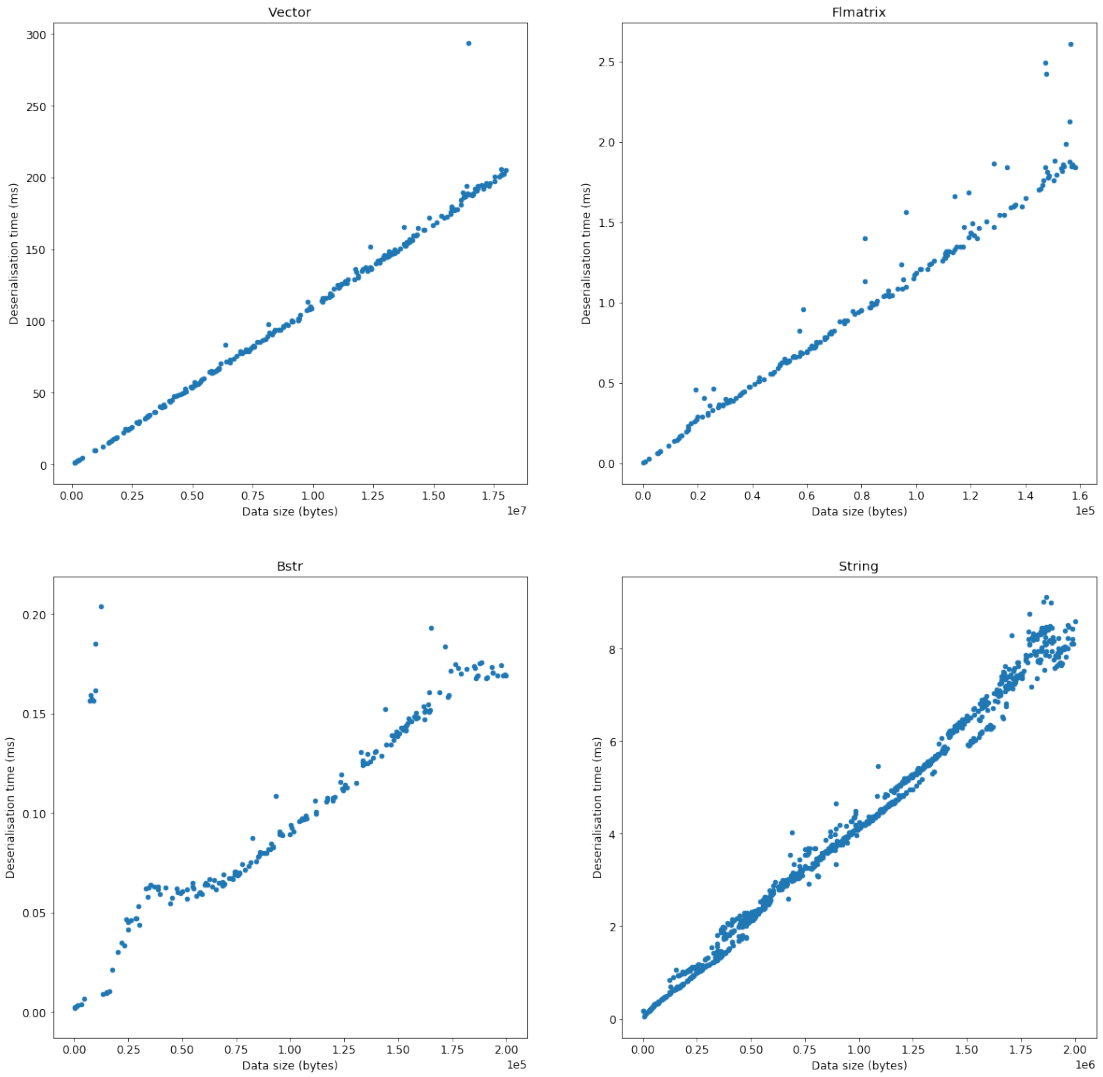

Figure B.4: Deserialisation Time against Data Size separated by type (Racket; FATA)

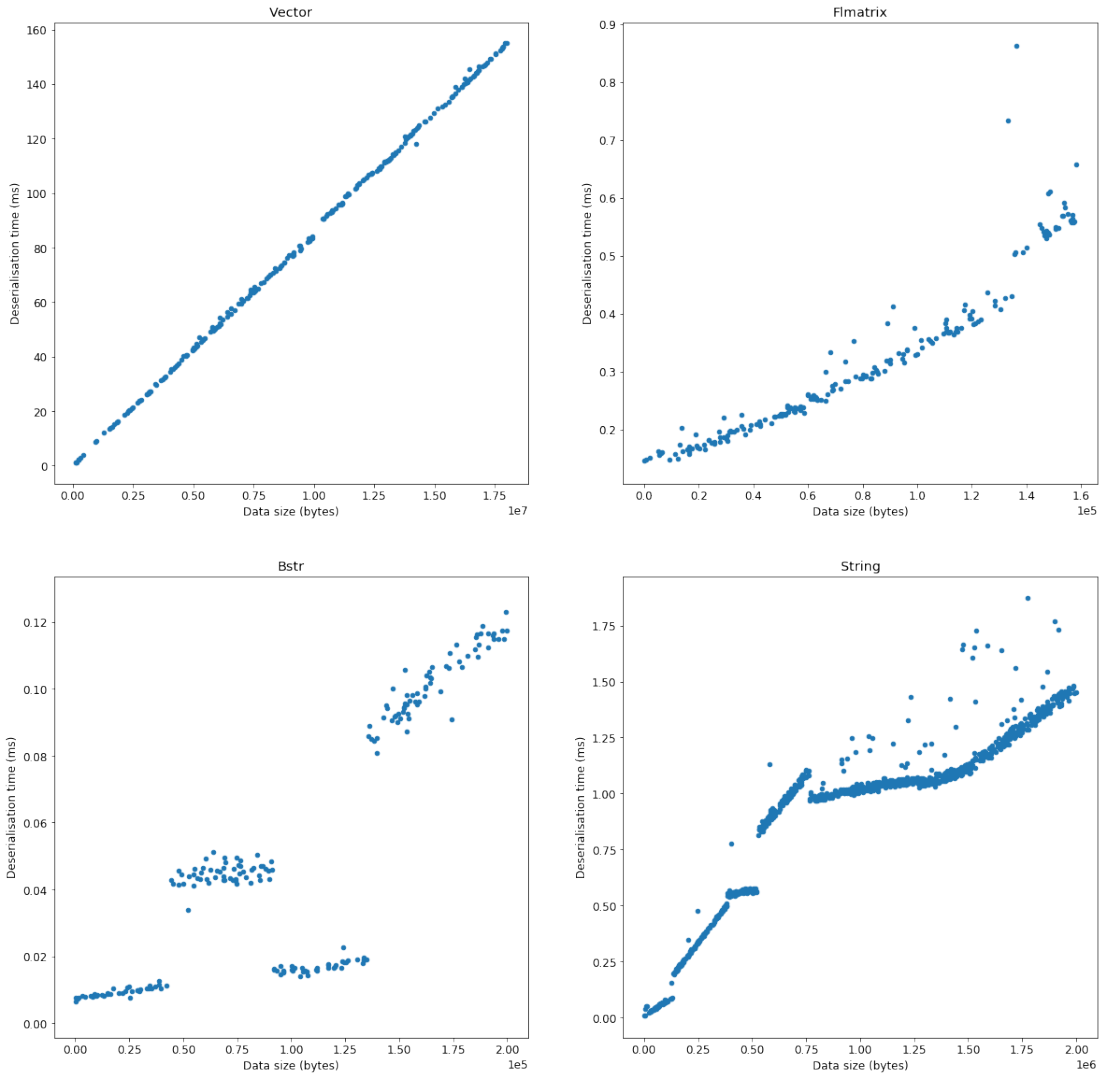

Figure B.5: Deserialisation Time against Data Size separated by type (Pycket; FATA)

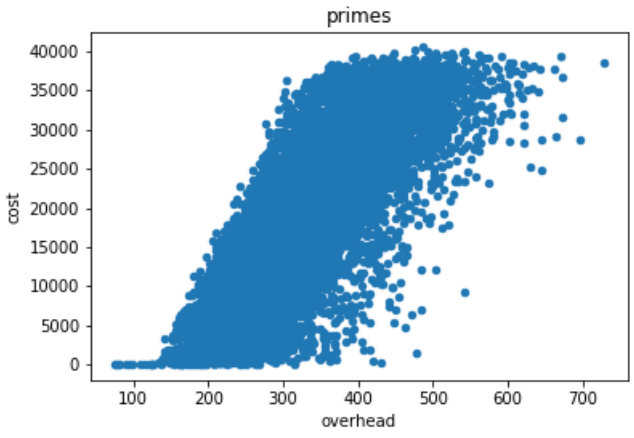

Figure B.6: Plot of predicted communications costs vs actual overheads for primes filter — GPG platform

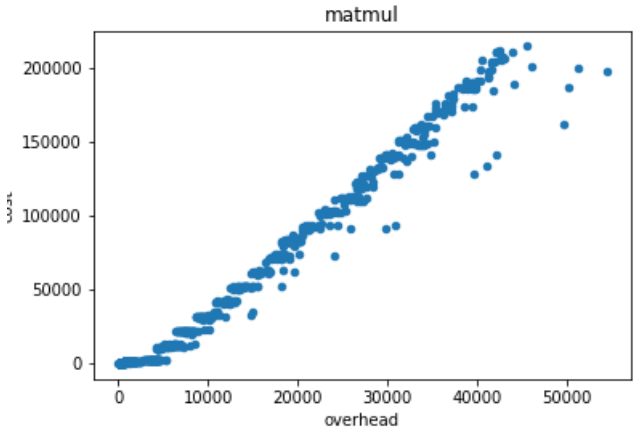

Figure B.7: Plot of predicted communications costs vs actual overheads for Matrix Multiplication — GPG platform

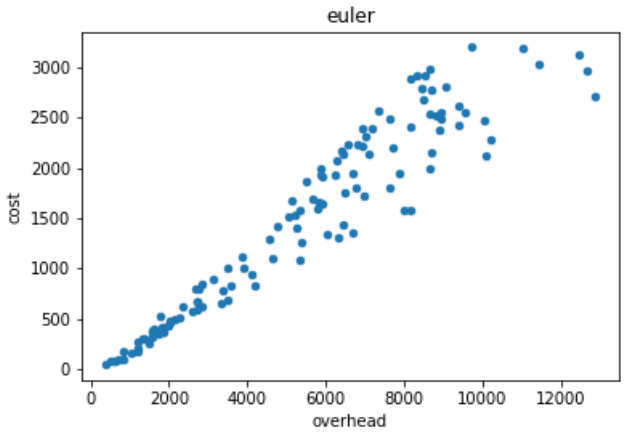

Figure B.8: Plot of predicted communications costs vs actual overheads for Sum Euler — GPG platform

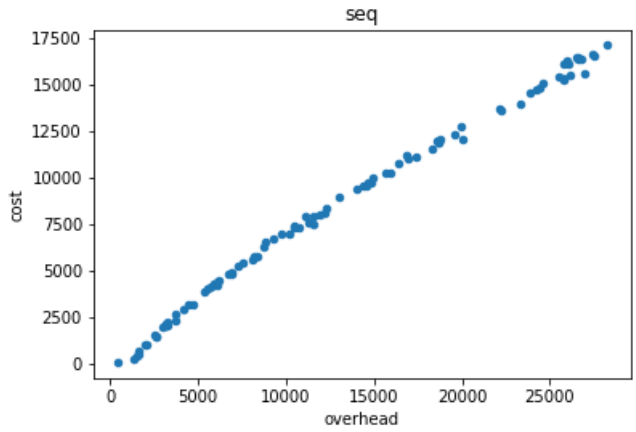

Figure B.9: Plot of predicted communications costs vs actual overheads for Sequence Align — GPG platform

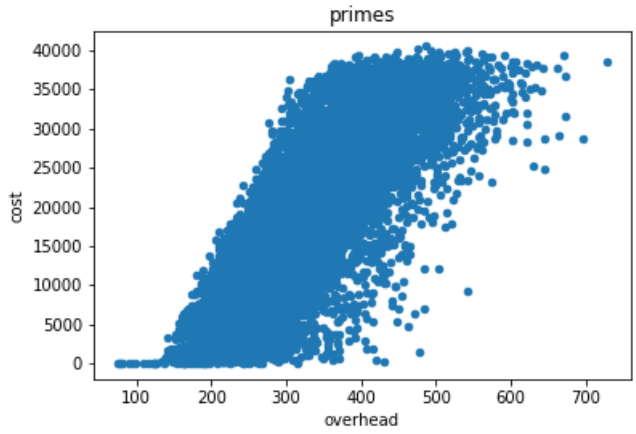

Figure B.10: Plot of predicted communications costs vs actual overheads for primes filter — FATA platform

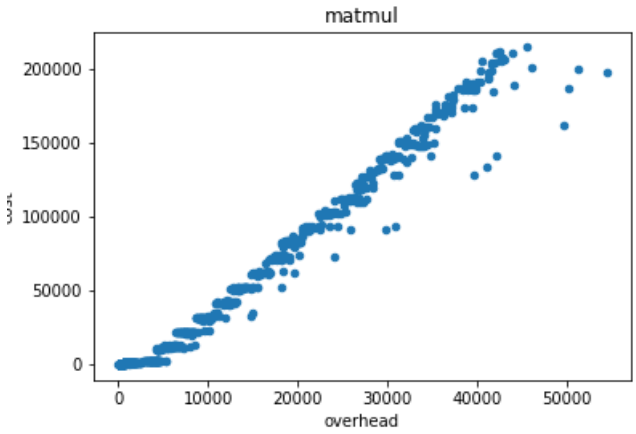

Figure B.11: Plot of predicted communications costs vs actual overheads for Matrix Multiplication — FATA platform

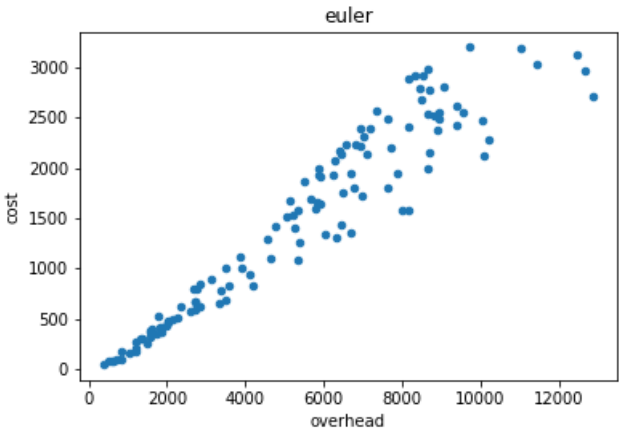

Figure B.12: Plot of predicted communications costs vs actual overheads for Sum Euler — FATA platform

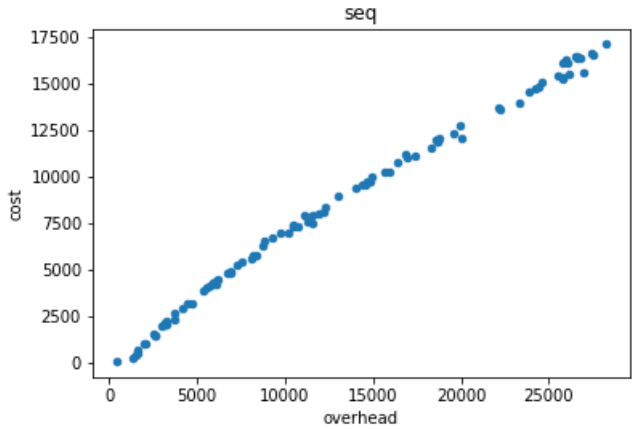

Figure B.13: Plot of predicted communications costs vs actual overheads for Sequence Align — FATA platform

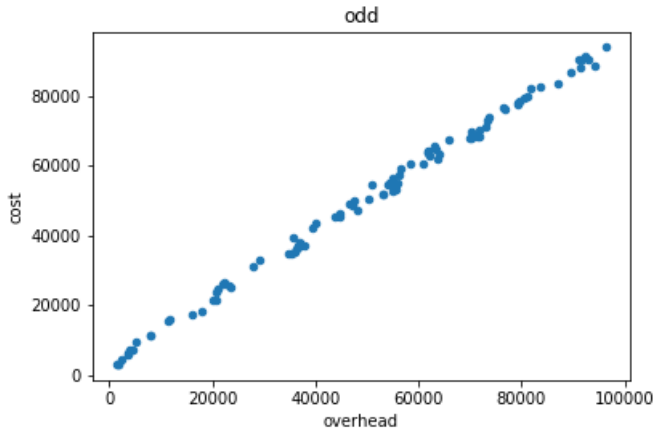

Figure B.14: Plot of predicted communications costs vs actual overheads for odd filter — FATA platform

Appendix C

Combined Cost Model

Chapter 5 contains a sample of the plots of time and various cost predictors against task group granularity. For space reasons the remaining plots are presented in this appendix. Figure C.1 is the only remaining GPG plot; Figures C.2 and C.5 contain the plots for the FATA platform.

C.1 Predicting Optimal Granularities

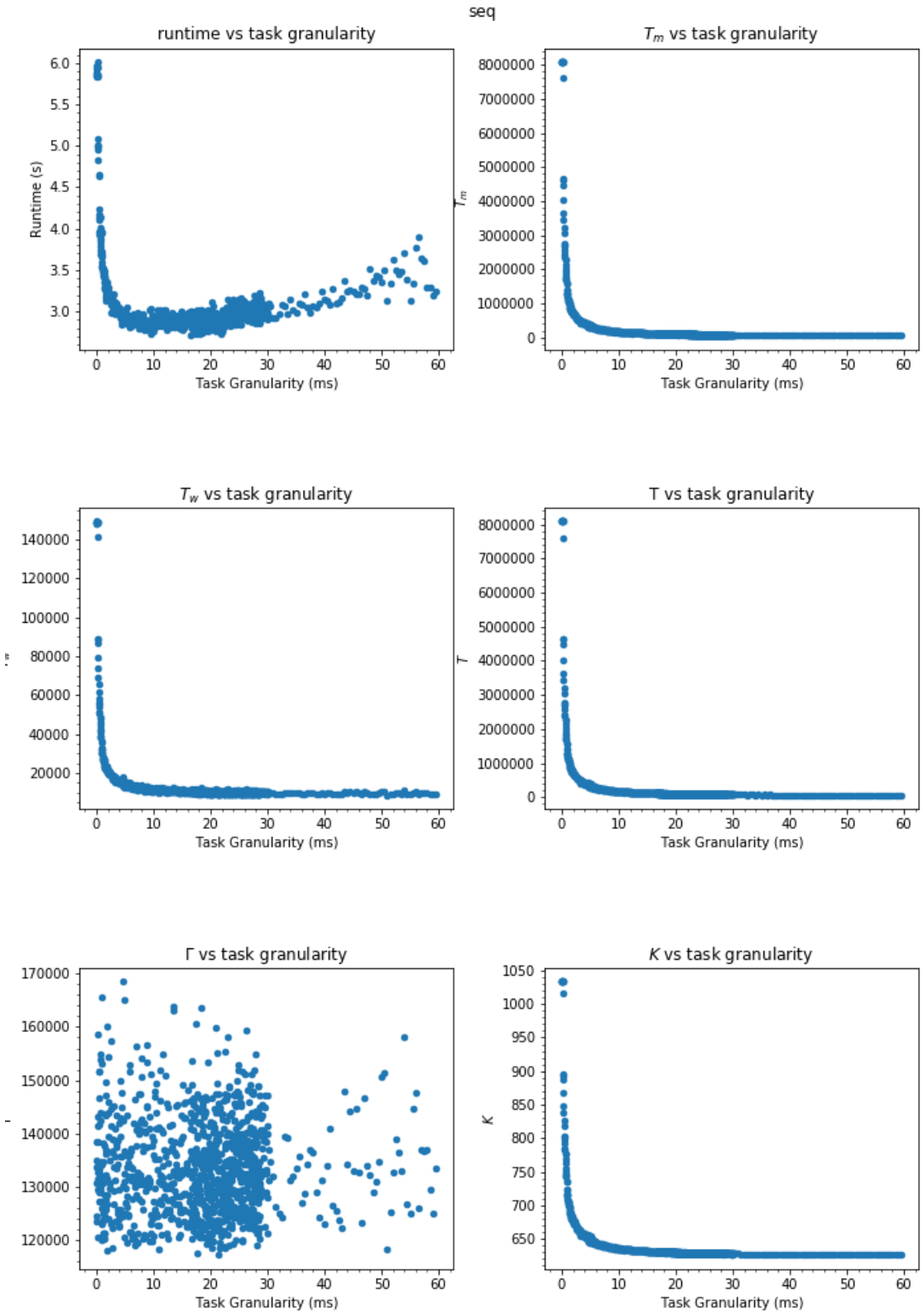

Figure C.1: Sequence Alignment Results — GPG

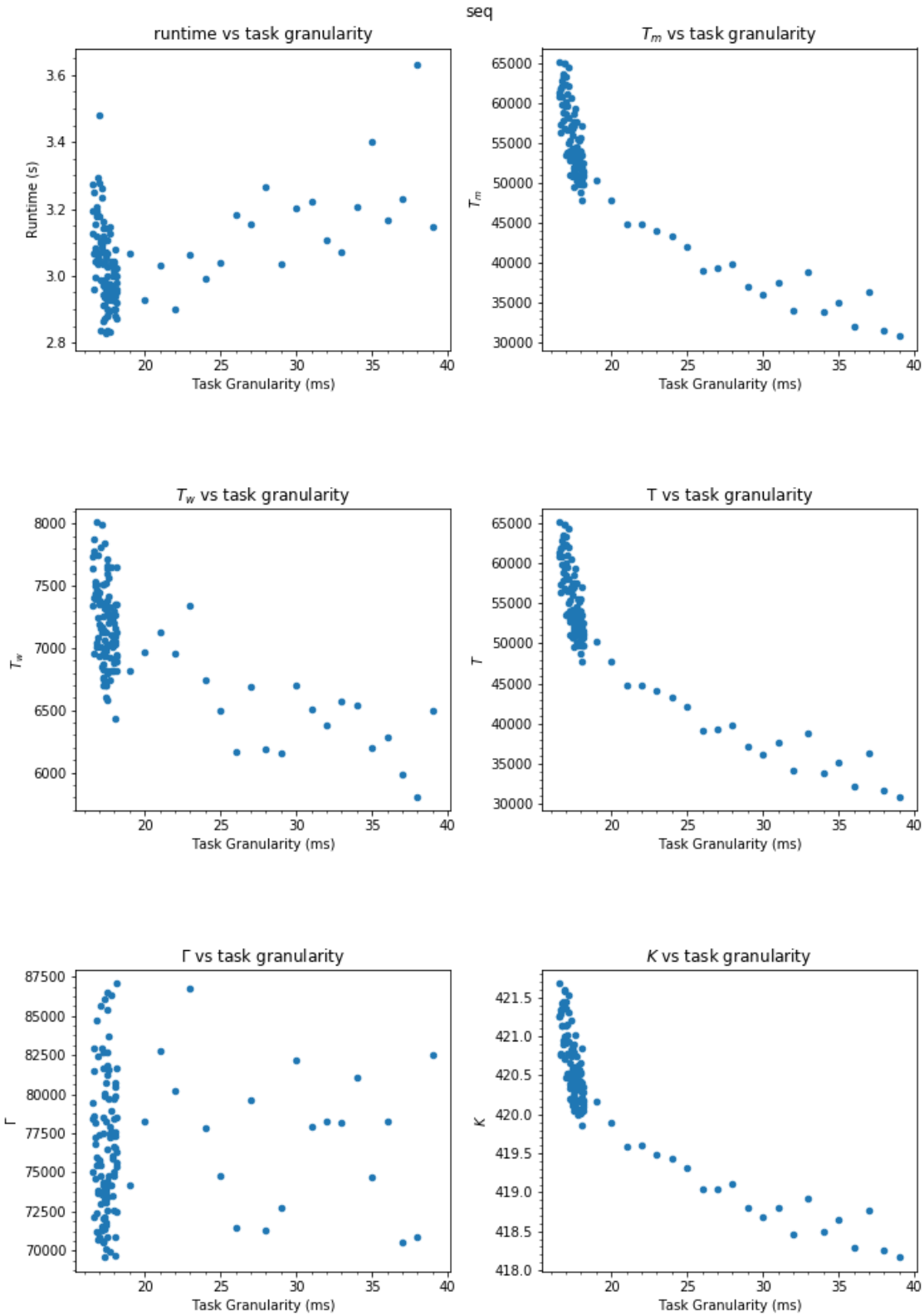

Figure C.2: Sequence Alignment Results — FATA

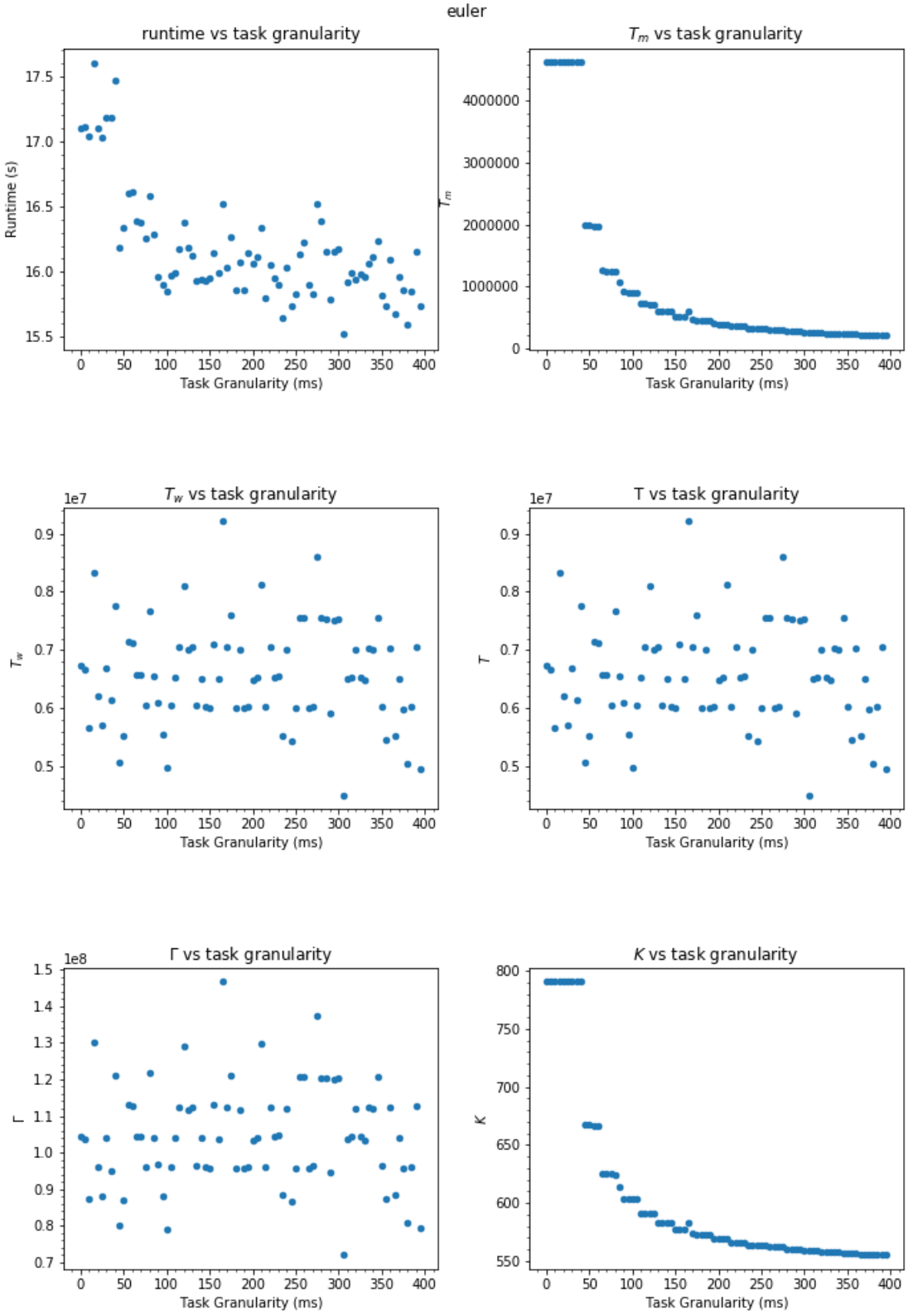

Figure C.3: Sum Euler Results — FATA

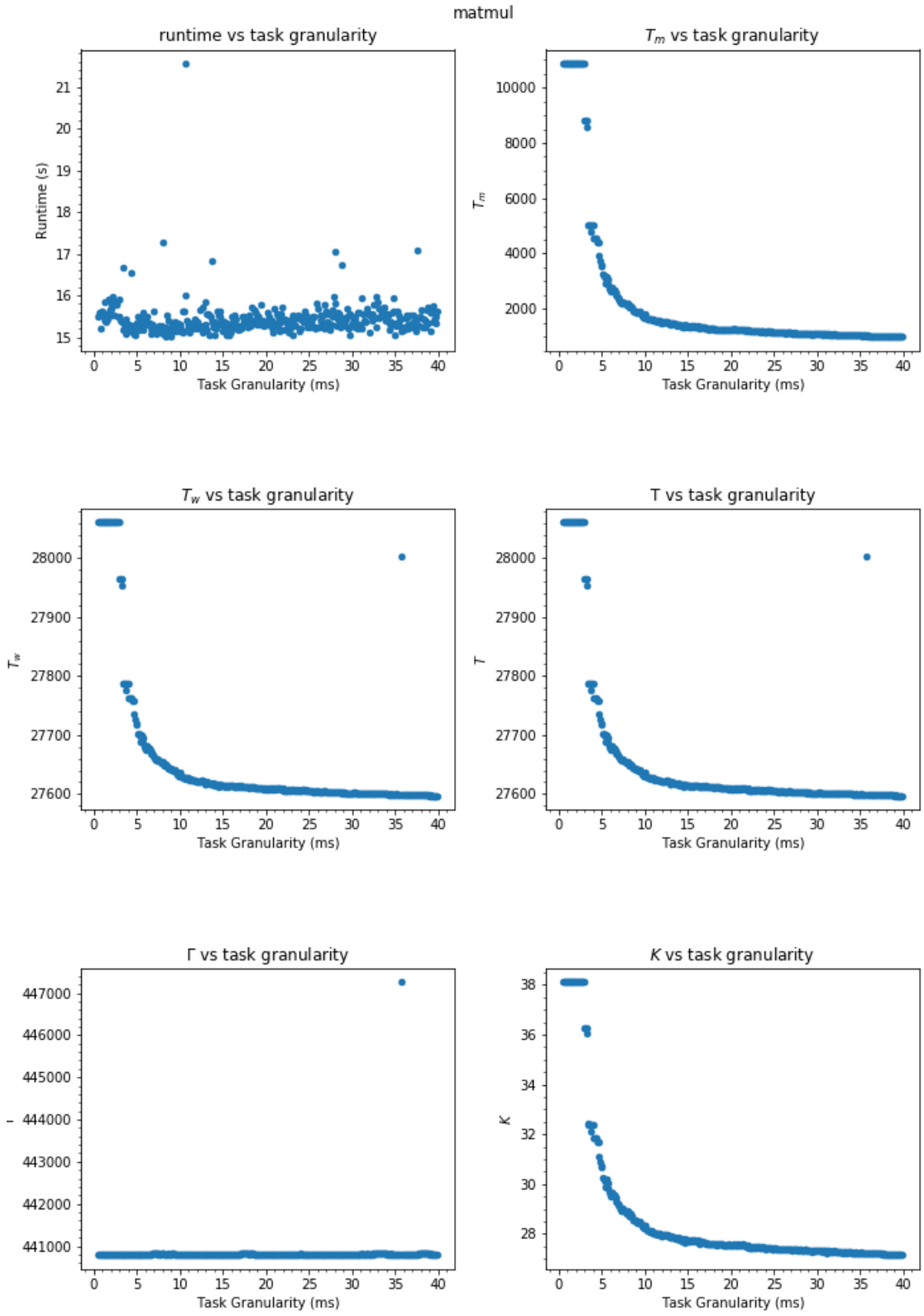

Figure C.4: Matrix Multiplication Results — FATA

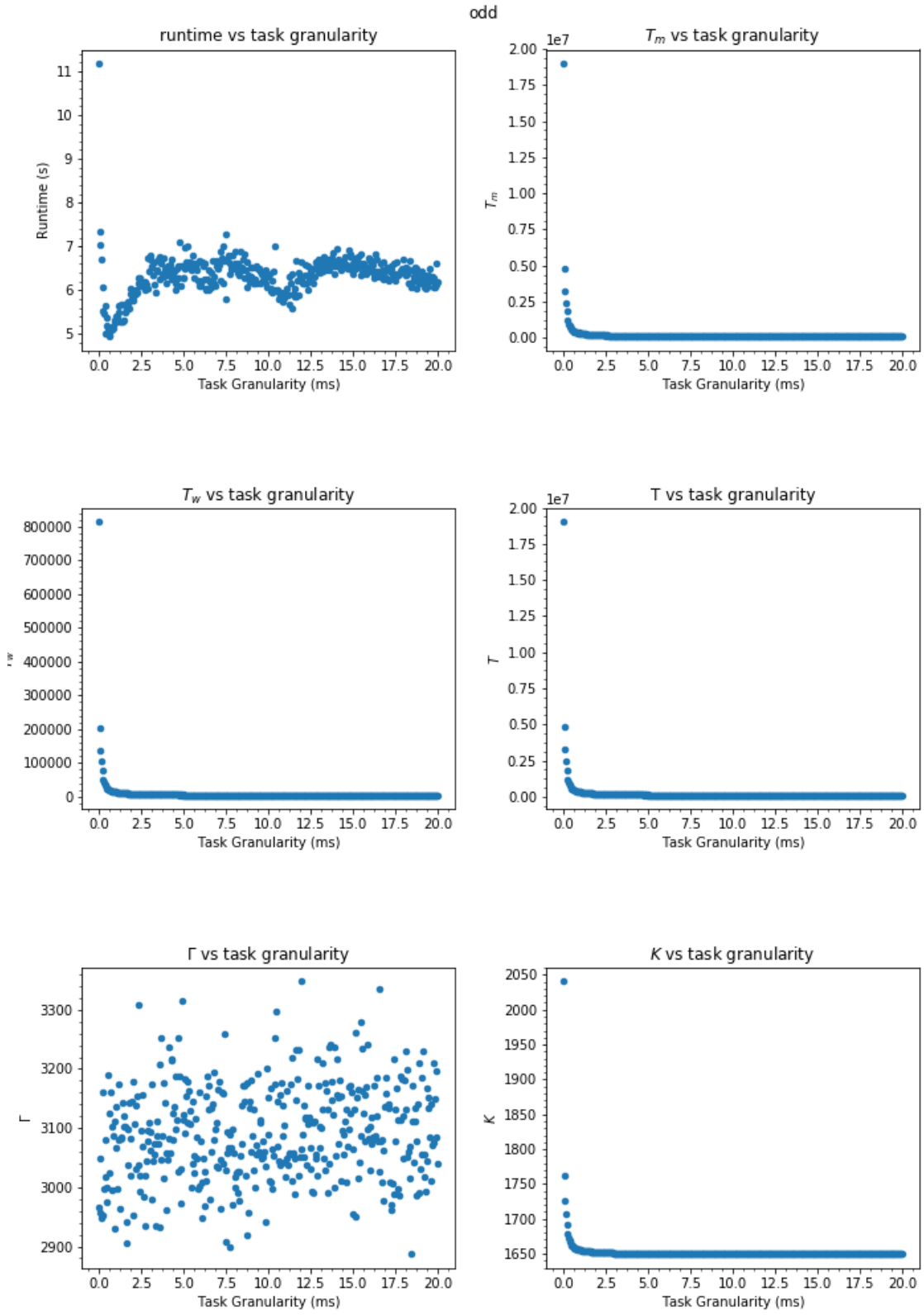

Figure C.5: Odd Filter Results — FATA