Elastic resource allocation for a structural design application

Emmanouil Giortamis

Thesis submitted in partial fulfillment of the requirements for the Masters’ of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Assistant Prof. Polyvios Pratikakis

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.
The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).
Elastic resource allocation for a structural design application

Thesis submitted by
Emmanouil Giortamis
in partial fulfillment of the requirements for the
Masters’ of Science degree in Computer Science

THESIS APPROVAL

Author:
Emmanouil Giortamis

Committee approvals:
Polyvios Pratikakis
Assistant Professor, Thesis Supervisor

ANGELOS BILAS
Angelos Bilas
Professor, Committee Member

KONSTANTINOS MAGOUTIS
Konstantinos Magoutis
Associate Professor, Committee Member

Departmental approval:
Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, June 2021
Elastic resource allocation for a structural design application

Abstract

Cloud computing is the on-demand delivery of computing services such as servers, storage, computing power, databases and networking, through the internet. Rather than owning their own infrastructure, individuals or companies can rent access to high-performant computing resources from a cloud service provider. Cloud providers typically offer pay-as-you-go pricing models which charge users only for the resources they use. However, applications tend to have varying resource demands depending on both incoming traffic rates and incoming workload types. Resource over-allocation leads to wasted resources and thus, money, while under-allocation leads to Service-Level-Objective (SLO) violations. To this end, cloud computing platforms adopt horizontal and vertical elasticity in order to timely scale the application’s resources on demand. Horizontal elasticity replicates the application’s resources while vertical elasticity resizes them. It came to our attention that both industry and scientific literature focus more on horizontal elasticity than on vertical elasticity. Vertically elastic resource scaling is essential for applications with workload-dependent and spiky resource demands, however.

In this thesis we present a vertically elastic resource allocator for fine-grained CPU-time allocation. Our proposed algorithm targets applications with job dependent parallelization spikes and accounts for variable traffic rates. Our example application is a Greek commercial structural design application used by civil engineers, named RAF. Its back-end, RAF::Solver, computes a building’s static analyses by solving linear algebra equations and factorizing matrices using parallel Cholesky decomposition. Part of our work was to port the RAF::Solver to Linux, containerize, and deploy it as a cloud service. Then, our methodical profiling and benchmark analysis showed that each RAF::Solver instance has different parallelization speedup margins and thus CPU demands, due to each building’s unique properties. Based on these observations we implemented both static and elastic CPU-time allocation schemes. Our evaluation analysis indicates that our fine-grained, vertically elastic CPU-time allocator yields better parallelization exploitation, up to 77% higher resource utilization and up to x10 less SLO violations, compared to the static allocation approaches.
Ελαστική διανομή πόρων για εφαρμογή στατικής ανάλυσης κτιρίων

Περίληψη

Ο υπολογισμός νέφους παρέχει υπολογιστικές υπηρεσίες όπως εξυπηρετήσεις, αποθήκευση χώρου, υπολογιστική ισχύ, βάσεις δεδομένων και δικτύωση, μέσω του διαδικτύου, κατ' απαίτηση. Αντι να αγοράζουν τις δικές τους υποδομές και τα δικά τους υπολογιστικά συστήματα, ιδιώτες και εταιρείες έχουν την δυνατότητα να ενοικιάζουν υπολογιστικούς πόρους ψηλής απόδοσης από παρόχους υπολογισμού νέφους. Αυτοί οι πάροχοι συνήθως προσφέρουν μοντέλα κόστους τύπου 'πλήρωσε όσο χρησιμοποιείς', τα οποία χρεώνουν τους χρήστες μόνο για τους πόρους που χρησιμοποιούν.

Όμως, οι εφαρμογές τείνουν να έχουν μεταβλητές απαιτήσεις σε πόρους κατά τη διάρκεια εκτέλεσής τους για δύο λόγους: μεταβλητά εισερχόμενα ποσοστά κυκλοφορίας και διαφορετικά τύποι των εισερχόμενων εργασιών. Η διανομή περιττών πόρων οδηγεί στην μη-αξιοποίηση τους και στην σπατάλη χρημάτων, ενώ η διανομή λιγότερων από τους αναγκαίους πόρους οδηγεί σε παραβιάσεις συμφωνιών επιπέδου υπηρεσίας. Γι' αυτό τον λόγο, οι εκπομπές υπολογισμού νέφους εφαρμοζούν τεχνικές ελαστικότητας προκειμένου να κλιμακώνουν τους πόρους μιας εφαρμογής σύμφωνα με τις απαιτήσεις της. Οριζόντια ελαστικότητα σημαίνει προσθήκη περισσότερων πόρων προκειμένου να εκτελεστούν περισσότερες εφαρμογές παράλληλα, ενώ κάθετη σημαίνει μεταβίβαση εργασιών από πόρους προκειμένου να έχουν περισσότερους οι εφαρμογές που έχει εκτελεσθεί. Τόσο στην πραγματική ζωή και στην ακαδημαϊκή περιόδο των πάροχων πόρων, καθώς και στην πραγματική περίπτωση, η εκμετάλλευση των πόρων επιτύχεται μετά από μεταβλητά εισερχόμενα ποσοστά κυκλοφορίας και διαφορετικές απαιτήσεις σε πόρους.

Σε αυτή την εργασία παρουσιάζουμε έναν κάθετα ελαστικό διανομέα πόρων που διανέμει χρόνο των επεξεργαστών με υψηλή ακρίβεια. Απευθύνεται σε εφαρμογές των οποίων ο παραλληλισμός είναι πολυποίκιλος και εξαρτάται από τον τύπο των εργασιών και δεδομένων που έρχονται. Εφαρμόσαμε τον διανομέα σε μια Ελληνική εφαρμογή στατικής ανάλυσης ονοματεπώντας ΡΑΦ, η οποία χρησιμοποιείται από πολιτικούς μηχανικούς. Στην πραγματική περίπτωση, ο επιτυχής αναπόληπτης διαχείρισης των κτιρίων υπάρχει μεταβίβασης παράλληλου υπολογιστή Cholesky. Μέρος της εργασίας ήταν να διεξάγουμε επαναληπτικά τον επαναληπτικό παραλληλισμό της εφαρμογής ΡΑΦ και να εξετάσουμε την εκμετάλλευση των πόρων που διανέμει χρόνο των επεξεργαστών με υψηλή ακρίβεια. Απευθύνεται σε εφαρμογές των οποίων ο παραλληλισμός είναι πολυποίκιλος και εξαρτάται από τον τύπο των εργασιών και δεδομένων που έρχονται. Εφαρμόσαμε τον διανομέα σε μια Ελληνική εφαρμογή στατικής ανάλυσης κτιρίων, η οποία χρησιμοποιείται από πολιτικούς μηχανικούς. Στην πραγματική περίπτωση, ο επιτυχής αναπόληπτης διαχείρισης των κτιρίων υπάρχει μεταβίβασης παράλληλου υπολογιστή Cholesky. Γενικά, η εφαρμογή ίσως να διεξάγεται μέσω της εφαρμογής ΡΑΦ, η οποία χρησιμοποιείται από πολιτικούς μηχανικούς. Στην πραγματική περίπτωση, ο επιτυχής αναπόληπτης διαχείρισης των κτιρίων υπάρχει μεταβίβασης παράλληλου υπολογιστή Cholesky. Γενικά, η εφαρμογή ίσως να διεξάγεται μέσω της εφαρμογής ΡΑΦ, η οποία χρησιμοποιείται από πολιτικούς μηχανικούς.
ο υψηλής ακρίβειας και κάθετα ελαστικός διανομέας μας οδηγεί σε μεγαλύτερη εκ
- 
μετάλλευση του παραλληλισμού, αποδοτικότερη αξιοποίηση των πόρων έως και 77%,
και έως 10 φορές λιγότερες παραβιάσεις συμφωνιών επιπέδου υπηρεσίας, συγκριτικά
με τους στατικούς διανομείς.
Acknowledgements

First of all, I would like to thank my advisor, Professor Polyvios Pratikakis, for his help, support and guidance throughout this thesis. He proposed the main idea of this thesis and his broad knowledge on topics even outside his research interests has been of great value to me. I am also grateful to Professor Angelos Bilas and Professor Konstantinos Magoutis, for accepting to serve in my thesis oral examination committee.

My sincere thanks to Mr. Manos Tsagkarakis, co-founder and senior staff member of TOL, for his willing help and patience regarding the RAF application and its inner workings. I am also grateful to my friend, colleague and TOL staff member as well, Myron Tsatsarakis, for his excellent collaboration and valuable help during this work.

Special thanks to the members of the CARV laboratory of ICS-FORTH and especially to George Zervas, for his repeated guidance and support on Docker and Kubernetes topics.

Last but not least, I would like to thank my friends and family for mentally supporting me, their never-ending patience, encouragement and trust in me. I will always be grateful for Dimitra’s understanding and support during these last, stressful and difficult phases of this thesis.
Contents

Table of Contents i
List of Tables iii
List of Figures v

1 Introduction 1

2 Background 5
  2.1 The RAF structural design application . . . . . . . . . . . . . . . . 5
  2.2 RAF::Solver and buildings’ model . . . . . . . . . . . . . . . . . . 5
    2.2.1 Buildings’ Components . . . . . . . . . . . . . . . . . . . . 7
    2.2.2 XML representation . . . . . . . . . . . . . . . . . . . . . 8
    2.2.3 Cholesky Decomposition . . . . . . . . . . . . . . . . . . . . 8
  2.3 Docker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  2.4 Kubernetes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3 Implementation 11
  3.1 Linux Porting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  3.2 Dockerization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  3.3 Input files generator . . . . . . . . . . . . . . . . . . . . . . . . . 13
  3.4 Profiling and benchmarking . . . . . . . . . . . . . . . . . . . . . 16
  3.5 Data-set creation . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
  3.6 Runtime prediction . . . . . . . . . . . . . . . . . . . . . . . . . . 23
    3.6.1 Linear Regression Models . . . . . . . . . . . . . . . . . . . 24
      3.6.1.1 Simple Linear Regression . . . . . . . . . . . . . . 24
      3.6.1.2 Ridge . . . . . . . . . . . . . . . . . . . . . . . . . 24
      3.6.1.3 LASSO . . . . . . . . . . . . . . . . . . . . . . . . 25
      3.6.1.4 ElasticNet . . . . . . . . . . . . . . . . . . . . . . 26
    3.6.2 Polynomial Regression . . . . . . . . . . . . . . . . . . . . . 26
    3.6.3 Support Vector Machine Regression . . . . . . . . . . . . . 27
    3.6.4 Random forest Regression . . . . . . . . . . . . . . . . . . . 27
  3.7 Baseline Resource Allocators . . . . . . . . . . . . . . . . . . . . . 28
    3.7.1 Static resource allocation . . . . . . . . . . . . . . . . . . . 28
List of Tables
# List of Figures

2.1 RAF’s User Interface ........................................... 6  
2.2 RAF’s Desktop ......................................................... 6  
2.3 XML representation of a building’s components ................ 9  

3.1 100 Nodes (red) and 50 Beams (green). No Node is not connected to every (or any) neighbor Node. .................. 14  
3.2 Fully Connected Nodes with Beams and Columns. ........... 15  
3.3 Fully Connected Nodes with Shells. .............................. 15  
3.4 Components’ impact on file size and runtime ................ 17  
3.5 Component’s impact on speedup ................................. 18  
3.6 FC(B/S): Fully Connected Beams/Shells ....................... 19  
3.7 Node connectivity comparisons ................................. 20  
3.8 Burst duration as percentage of the total runtime .......... 21  
3.9 Nodes’/Beams’ values cover most of the input domain’s range ... 22  
3.10 Shells’/Restraints’ values cover most of the input domain’s range . 22  
3.11 File size frequency distribution ............................... 23  
3.12 Different expected runtime for different number of threads (1 vs 4) 23  
3.13 Linear Regression with fit intercept and normalize options ... 24  
3.14 Ridge Regression Accuracy ............................... 25  
3.15 LASSO Regression Accuracy ........................................ 25  
3.16 ElasticNet Regression Accuracy ................................. 26  
3.17 Polynomial Regression Accuracy ............................... 26  
3.18 SVM Regression Accuracy ................................... 27  
3.19 Random Forest Regression Accuracy .......................... 27  
3.20 Baseline allocators’ design ....................................... 29  
3.21 YAML template .................................................... 30  

4.1 Baseline vs elastic allocation: Average job completion time .... 39  
4.2 Average completion time breakdown: Execution time (left) & waiting time (right) ........................................ 40  
4.3 Maximum waiting times (left), Waiting times > 60s percentages (right) ...................................................... 41  
4.4 Baseline vs elastic allocation: Average waiting time .......... 42  
4.5 Baseline vs elastic allocation: Average job completion time .... 42
Chapter 1

Introduction

Cloud computing is the on-demand delivery of computing services such as servers, storage, computing power, databases and networking, through the internet. Public cloud platforms ([3], [4], [8], [6], [5]) provide these services to both individuals and enterprises for a fee, typically offering a pay-as-you-go pricing model. On the contrary, private cloud providers usually accommodate specific organizations only. There are hybrid options as well, where elements of both public and private cloud computing are combined. Instead of building, buying, owning and maintaining their own infrastructures, individuals, companies and organizations can rent access to computing resources tailored to their needs. There are mainly three cloud computing types:

- Infrastructure-as-a-service (IaaS): Delivers physical resources such as servers, networking, storage and raw computing power. We can think of it as the lowest level of cloud services and tend to be the least expensive one.

- Platform-as-a-service (PaaS) is considered as the next level of the cloud stack. It usually provides developers with tools, services and workflows in order to create, develop, deploy and bundle software services, without worrying about the underlying infrastructure management.

- Software-as-a-service (SaaS) can be thought of as the highest level of the cloud stack. This type of service provides software licensing to customers with a pay-as-you-go pricing model or a monthly/yearly subscription. A SaaS example is Microsoft Office 365.

Cloud computing offers such advantages that people from all sectors were attracted in the last decade, leading to an explosion of both demand and provision of cloud services. Such advantages include and are not limited to: Using software from any device either by apps or browsers, accessing personal or company data and storage from anywhere, back-up of the aforementioned data and access to a broad range of tools. More importantly, developers can create and deploy applications or services on a global scale in a fast, easy and automated fashion.
In all three cloud computing types, applications and services rarely run on bare metal. Instead, Virtual Machine (VM) and Containerization technologies abstract the underlying physical machine. Virtual Machines usually do not access the underlying resources by themselves either. Instead, a software called Hypervisor abstracts the machine’s resources into pools and assigns them to multiple VMs, if needed, enabling them to run on the same host machine. Containers are a complementary virtualization method. They can be described as bundles of software that can run on top of both bare metal and VMs. Usually, they virtualize the whole operating system and are able to run anywhere, from datacenters to personal computers, the cloud and even mobile devices.

It is worth noting that most applications and services tend to have varying resource demands across their lifetime. Peak-levels of business activity or even flash crowds, idle time periods, and generally varying traffic rates as a function of time, all lead to varying resource demands. Moreover, depending on the application type, even the incoming workload types can lead to vast differences in resource consumption. Heavy workloads such as data analytics, machine learning, weather prediction and generally high-performance-computing types of incoming work, have completely different resource demands across different problems and/or different application instances. Without mechanisms that account for this variability when allocating resources to applications, resource over/under provisioning is unavoidable. Maintaining resources for the peak-traffic or peak computational demands is costly, under-utilizes and wastes resources. On the other hand, under-provisioning can lead to more subtle implications: By maintaining insufficient resources for the peak demands, Service-Level-Agreement / Objective (SLA/O) violations can occur. More specifically, applications and services can starve due to lack of resources such as network, memory or processing power. This can happen because of both high traffic demands and resource-intensive work. This starvation leads to lower quality of service and can often be observed by the application’s customers, causing their dissatisfaction.

To address this issue, cloud providers adopt and implement some kind of elasticity. Elasticity can be defined in many ways but we adopt (and quote) the definition presented in [14]:

(Elasticity is) “The degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.”

There are mainly two elasticity types regarding the cloud environment: horizontal and vertical. Horizontal elasticity consists of either replicating an application’s resources or replicating the application itself. For example, adding more nodes in a cluster or booting more application instances are types of horizontal elasticity. On the other hand, vertical elasticity consists of resizing an application’s existing resources such as CPU time, memory, network etc. For example, increasing the
memory capacity and allocating more CPUs for an application is one kind of vertical elasticity.

However, we noticed that not only is substantially more work on horizontal elasticity compared to vertical, but also elasticity regarding VMs compared to Containers. There are a few reasons for this: At first, resource allocation and job scheduling are NP-Complete, open problems and very hard to generalize in most cases. Moreover, VMs are a significantly more mature technology, and thus, studied, while containers where quite recently (widely) adopted. Additionally, horizontal elasticity is often more convenient to implement and adopt. For example, long running web services which serve client requests can easily be replicated in order to serve more clients simultaneously, as traffic increases. On the contrary, there are less contained applications and services that benefit from a specific resource scaling without having other bottlenecks, thus requesting horizontal scaling anyways. Finally, even when vertical elasticity for containers is addressed, it is coarse-grained or targets long-running services and applications. As far as we know, there are very few studies that target fine-grained, vertically elastic resource allocation for applications whose resource demands are mostly workload-type-dependent.

A partial solution to the aforementioned allocation problem was implemented while working on an application that fits the above criteria: Each application instance is short-lived; it executes a specific task and then terminates. Moreover, each instance has completely different (CPU) resource demands depending on the task's configuration and properties, due to the underlying parallelization optimizations. More importantly, however, not all resource demands should be accommodated since the parallelization speedup gain might not be worth it (for the amount of resources assigned). Finally, for the reasons we described earlier in this section, both the application’s distributors and users could benefit from its deployment as a service on the cloud.

In this work, we ported to Linux, containerized and deployed as a service part of a structural design application, named RAF. This application is used by civil engineers in Greece for static and anti-seismic analyses. It consists of various sub-applications with their respective front-end (User Interface), each serving a different civil engineering demand. Our focus was on RAF::Solver, the back-end of the main component which is the module responsible for the whole static analysis and building stability/durability. The lack of enough input available drove us to implement an automatic input generator. Using this generator we carefully created realistic datasets on scale which we used in the next parts of this work. Then, we applied methodical and extensive benchmarking and profiling techniques in order to determine the RAF::Solver’s runtime behaviour and resource demands. We explored the Solver input’s impact on the runtime and parallelization speedup. More specifically, we had interesting findings regarding the serial and parallel portion of the runtime, their respective start and duration, and lastly, parallelization speedup as a function of threads and building’s configuration. This analysis showed that patterns exists and all the above metrics could be accurately predicted at priori. This was indeed the case since we trained various regression models, achieving up
to 98% accuracy. Finally, we implemented a fine-grained, vertically elastic resource allocator for workload-dependent parallelization and thus, (virtual) CPU resource demands. Our work currently addresses virtual CPU allocation only but can be easily extended to other resources such as memory, storage and even network. Our methodical evaluation showed that an elastic resource allocator that accounts for both parallelization patterns and varying traffic rates can outperform static allocation policies even if they are not naive and are somewhat parallelization-aware. In summary, we make the following contributions:

- We deployed a state-of-the-art commercial structure application as a service using Docker containers and Kubernetes.

- We propose a methodology/pipeline for the implementation of vertically elastic resource allocators custom-tailored to each application’s resource demands.

- Our extended profiling and benchmarks gave a thorough insight on the application’s runtime behaviour and resource demands. This information can help its developers optimize specific components and exploit parallelization more efficiently.

- We propose a fine-grained, vertically elastic CPU allocation scheme for applications with problem-dependent parallelization speedup patterns. The allocator factors both variable traffic rates and problem-specific resource demands.

The rest of this thesis is organized as follows. We first give the necessary background information on the RAF::Solver application, Docker and Kubernetes. In Chapter 3 we provide a detailed description of our implementation of the above contributions, along with some necessary intermediate steps to achieve them. Then, Chapter 4 contains our in-depth evaluation and results. In Chapter 5 we present the relevant and related work. Finally, we conclude and discuss possible future work in Chapter 7.
Chapter 2

Background

2.1 The RAF structural design application

RAF [23] is a commercial, state-of-the-art structural design and engineering application developed and distributed by TOL [24]. Civil engineers use this application for static and anti-seismic analysis of buildings and constructions. It is developed and targeted on/for the Windows platform and follows the Microsoft-Office user interface, as shown in Figures 2.2 & 2.1. RAF consists of several components, each specializing to different civil engineering demands. Some example components are:

- Main RAF component: manages all over components and executes static and/or anti-seismic analyses
- Reinforced concrete: Computes the reinforced-concrete-based buildings’ durability checks
- Foundation: Simulates, computes, checks and designs specific building foundations and bases
- Reinforcement: Evaluates the reinforced buildings’ bearing capacity
- Metal building: Checks and evaluates the durability of steel buildings

Since the main component is responsible for the whole building’s static analysis, our work targets its back-end and more specifically, the core RAF::Solver sub-component.

2.2 RAF::Solver and buildings’ model

The RAF::Solver is responsible for solving linear algebra equations of the following form:

$$\vec{F} = K\vec{U}$$
CHAPTER 2. BACKGROUND

Figure 2.1: RAF’s User Interface

Figure 2.2: RAF’s Desktop
where $\vec{F}$ is the vector of applied forces on the building’s components, $K$ is the degrees of freedom matrix (which describes each component’s (in)ability to move) and $\vec{U}$ is the vector of motions (in which direction, if any, each component is going to move). The static analysis’ aim is to compute $\vec{U}$ and determine if those movements can destroy or damage the building or any of its components. Both $\vec{F}$ and $K$ are known since they are the Solver’s input (given by the civil engineer). Thus, the equation must be solved for $\vec{U}$:

$$\vec{F} = K\vec{U} \Rightarrow K^{-1}\vec{F} = K\vec{U} \Rightarrow K^{-1}\vec{F} = I\vec{U} \Rightarrow \vec{U} = K^{-1}\vec{F}$$

where $I$ is the identity matrix and $K^{-1}$ is the inverse matrix of $K$. At this point, the only unknown variable is the $K^{-1}$ which is easier to compute versus the original $K$ matrix. For this purpose, Cholesky decomposition (2.2.3) is used.

### 2.2.1 Buildings’ Components

Modern buildings tend to have really complex design, structure and construction, which only professionals in relevant fields (e.g. civil engineers, architects etc.) can understand and describe in detail. However, some basic and really simple civil engineering terms are necessary in order to understand this work. The main reason behind this is that we need to understand the Solver’s behaviour in terms of computation complexity, runtime overheads, parallel-ization, speedup etc., which we extracted from careful analysis of benchmarks and extended profiling. Part of the profiling process was creating both building generator scripts, which composed possible building configurations to be solved, and whole data-sets of buildings/problems for parallel-ization patterns extraction. Therefore it was important to get a better insight into the Solver’s input.

The RAF::Solver accounts for many different building components but the main and most common ones, on which we will focus, are:

- **Storeys**: A floor/level abstraction. A building can have multiple storeys. Components (e.g. beams) may belong to different or the same storeys, depending on their position along the height ($z$) axis.

- **Nodes**: Simple points in space abstraction. A node is defined by its three space coordinates $x, y, z$.

- **Beams**: A connection between Nodes abstraction. There are various Beam types such as vertical (named columns) and horizontal ones, both abstracted as a line, wall-like ones abstracted as a plane etc. A Beam is defined by (among others) its start-to-end Node connections, number of segments and Beam type.

- **Shells**: A connection between Nodes abstraction as well. A shell is a surface/plane, part of a division-ed plate. Can be of triangular or rectangle
shape if connecting three or four nodes, respectively. A Shell is defined by (among others) its three/four Node connections.

- **Restraints**: A restriction on a Node’s movement and rotation along the three axes. For example a value of one on the $x$ axis means that the Node cannot move along the $x$ axis (but can rotate along it). A Restraint is defined by six binary values (three movements plus three rotations) and the Node’s identifier, on which it is applied to. A maximum of one Restraints can be applied to a Node.

- **Node Mass**: Part of a Node’s load, along with Node Forces. It abstracts the mass concentrated at a certain component of the building. A Node Mass is defined by the Node’s identifier it is applied to and three values, the three-dimensional position of the mass. A Node can have a maximum of one Node Mass elements.

- **Node Force**: It abstracts a force applied to a certain point/component of the building. A Node Force is defined the Node’s identifier it is applied to and three values, the three-dimensional strength (in newtons) of the force.

- **Beam Distributed Load**: A load applied to the whole or part of a Beam’s length. It is defined by the Beam’s identifier it is applied to and values indicating which part of the Beam has a force of given strength applied to it.

2.2.2 XML representation

The Solver expects an XML file as input in order to compute the described building’s static analysis. Such XML files describe in detail every piece of information the Solver expects, such as the analysis type (e.g. Beam or Shell only), the solving method, output information (e.g. verbose vs simple static analysis output), materials used in the building, and most importantly, the XML description of the building’s components described in the previous section. This file is parsed and all building information is copied in-memory as C++ objects (e.g. STL data structures) for faster access, avoiding file parsing at runtime. As we will see in later sections, this parsing and in-memory allocation of every building component plays an important role in the Solver’s runtime behaviour, especially when the building has a lot of components and thus, large XML files to describe it. An example XML is shown in Figure 2.3.

2.2.3 Cholesky Decomposition

Finally, after the XML file parsing has been completed, the Solver has to assemble the $\vec{F}$ vector and $K$ matrix based on the building’s components. Even slight differences between numbers of certain components, such as Restraints, lead to completely different degrees of freedom matrices and forces vectors. Moreover,
2.3. **Docker**

Docker [11] is a lightweight virtualization technology that allows to package an application with all of its dependencies in to one standardized unit for software deployment. It is an open source project that builds on many long-familiar technologies from operating systems research: LXContainers, virtualization of the OS, and a hash-based or git-like versioning and differencing system, among others. Docker uses a client-server architecture. There are four main internal components of docker, including Docker Client and Server, Docker Images, Docker Registries, and Docker Containers. Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

The official documentation (docs.docker.com) already provides a thorough description of the above, as well as an introduction in how to use Docker software;

---

**Figure 2.3:** XML representation of a building’s components

The Cholesky decomposition algorithm is not highly parallel-izable by itself (when running on CPUs), plus it is optimal for sparse matrix computations ([2], [21], [22]). Additionally, $K$ is stored in a skyline-storage fashion ([12]), which means that a Skyline Cholesky algorithm variant is required. This variant, however, is not so easily adapted for massively parallel computing. Thus, the Cholesky decomposition along with the (parallel) matrix/vector assembly parts, are the main and varying (between building configurations) parallelization speedup sources.
here our focus is on why we used Docker for our work.

- Dependencies: RAF depends on more than 20 libraries, including enterprise, commercial ones [7]. It is at least time-consuming if not impossible to resolve them on any possible system, with different package managers and versions' availability. Also, cloning those libraries’ repositories and compiling their source code is mundane enough (and mandatory) for two of the libraries RAF uses [15], [10].

- Portability: Applications built inside docker containers are extremely portable. Since RAF depends on various Fortran and C++ linear algebra libraries, it is of crucial importance that there are always the same (and inter-compatible) versions of them available.

- Scalability: Docker has the ability that it can be deployed in several physical servers, data servers, and cloud platforms. It can also be run on every Linux machine. Containers can easily be moved from a cloud environment to localhost and from there back to cloud again at a fast pace.

2.4 Kubernetes

Kubernetes [18] is a portable, extensible, open source container orchestration engine for automating deployment, scaling, and management of containerized applications, which facilitates both declarative configuration and automation. Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable. Kubernetes allows automatic various storage system mounts, such as local storages, public cloud providers, and more. We used this feature to mount our data-sets as a local volume inside RAF’s container. Another important feature is that you can provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You can tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources. Kubernetes mainly offers cluster and horizontal scalability/elasticity for dynamic resource allocation and resource scaling. Its vertical scaling features (5.2) were quite recently implemented and not fully developed/supported yet. There are many important and useful features to be added in the future which will help developers to properly scale their applications vertically.
Chapter 3

Implementation

3.1 Linux Porting

At the time of writing, the whole RAF application (including the Solver) was set up to be developed and built on Windows only, and more specifically, on Visual Studio [9] IDE. This Visual Studio setup process was tedious, time consuming and the library dependency resolving usually required help from someone within TOL. Additionally, the RAF::Solver consists of more than 40 C++ and 320 Fortran source code files which in turn contain more than 7,500 and 80,000 lines of code, respectively. As already mentioned in 2.3, the RAF::Solver depends on more than 20 non-standard libraries, many of which need to be installed and properly configured for the Solver to compile and run correctly. Thus, manually compiling and linking the source code on Linux was out of the question obviously.

It was beneficial for this work to not only port the Solver component to Linux (the rest components are irrelevant to this work), but automate the process as well. With a fast Linux porting mechanism, we could easily set up, test and benchmark the Solver’s behaviour and scalability on completely different environments, independently of specific IDEs. Despite being a time-consuming and tedious work as well, the porting process would be a one-time overhead. After that point, the Solver would be up-and-running in a matter of a few keystrokes.

The porting procedure consisted of the below steps:

1. Replace Windows-only C++ header files with cross-platform ones.
   E.g. substitute windows.h with the header files it includes internally.

2. Add missing header files which Visual Studio automatically included.

3. Change all file paths to cross-platform ones ( "..\.." to "../..").

4. Enable the cross-platform character encoding and widen/narrow methods.

5. Create a directive which defines attributes and/or directives depending on the underlying platform.
CHAPTER 3. IMPLEMENTATION

E.g. dynamic library symbol exporting for Linux:
\_attribute\_((visibility("default"))). For Windows: \_declspec(dllexport)

6. Change various Fortran Windows-only subtleties to cross-platform ones.
   E.g. arrays may need to have their elements’ type defined at declaration
time but not inferred.

7. Create Makefiles which compile and build each dynamic library (.so) and
   link them together into the complete RAF::Solver executable.

After a successful Linux porting we still needed an automated way to down-
load, install and configure the third-party library dependencies: Firstly, we make
sure that the desired building tools exist in the system, such as autoconf, gnupg,
csh, correct g++ version (9+) etc. Then, most of the library dependencies are re-
solved by pulling from each distro’s (default) package manager repositories, while
the rest are cloned from Github, automatically configured for RAF, and then com-
piled from source with their respective Makefiles. Finally, the proper PATH and
LD_LIBRARY_PATH must be set for the dynamic library linking at runtime. The
whole process, from zero to having a completely working executable, takes around
10 minutes of idle developing time (meaning there is no need for any input or work
by the developer).

We implemented two major ports: one for Ubuntu 18.04 LTS (plus 20.04 LTS,
which had almost no changes) and one for CentOS 7.

In conclusion, we have way more flexibility over the available machines and
platforms to test, benchmark, and evaluate the Solver and its scalability. In a
few minutes of idle time, the Solver was up-and-running on at least two local
Ubuntu machines, one Ubuntu virtual machine running on top of Windows, and
one CentOS server, part of a cloud cluster.

3.2 Dockerization

The main goal of this thesis was to containerize and deploy the RAF::Solver to
the cloud. We chose Docker as our container engine for the reasons described in
2.3 plus its big community and relative maturity. With a proper Linux porting
in our hands, the Dockerization process was quite simple and fast. The Solver’s
Dockerfile pulls an nvidia image from the Docker repository (named Hub) since the
Solver’s Cholesky library expects CUDA drivers to be installed in the system even
if no CUDA acceleration is utilized. This image basically is an Ubuntu 20.04 image
with CUDA drivers. The automation scripts described in the previous section are
copied and run into the image which completes the Solver’s setup into the nvidia
image. The Solver’s image can then be build with the docker build command
and run with the docker run command. An XML file is passed as argument to
the docker run command. The Solver’s results are stored in the NFS mounted
directory, which is the same it reads the XML files from.
3.3 Input files generator

In order to benchmark the Solver’s resource requirements, parallelization speedup, and train the runtime prediction models (3.6), it was important to have enough input available, i.e. a few thousands of input files. Unfortunately, this was not the case since the available input at the time was less than 20 XML files. Moreover, most of the files/problems were really small in terms of number of components, e.g. a few dozens of Node elements, Beams etc. This input’s main usage was for simple testing and debugging for which reasons we used them too, i.e. to check if the Solver works correctly after the Linux porting and the Dockerization process. More importantly however, the components’ range of values was not broad enough to cover (most of) the input domain, which of course could lead to unexplored parallel problems with great speedup margins. Finally, there was no automatic way to produce such files on scale without spending hours of manually filling in XML files with the variables and components needed for each test.

For this purpose, we implemented an automatic input files generator. It consists of a fairly simple Python script that uses an empty XML file and fills in the variables/components needed according to the script’s console input. An example invocation of the script could be:

```
python generator.py – n 100 – b 100 – r 100
```

which would generate an XML input file with 100 Node, 100 Beam and 100 Restraint, components.

The script supports the following options, which are most of the Solver’s major components:

- Number of Storeys
- Number of Node components (per Storey). Must be a natural perfect square, for example: 16, 81, 100, 225 etc.
- Number of Beam components (only in one version). Upper limit:

\[
\text{Storeys} \times 2 \times \sqrt[3]{\text{Nodes}} \times (\sqrt[3]{\text{Nodes}} - 1)
\]

- Number of Column components (only in one version). Upper limit:

\[
\text{Nodes} \times (\text{Storeys} - 1)
\]

- Number of Shell components (only in one version). Upper limit:

\[
((\sqrt[3]{\text{Nodes}} - 1)^2 \times \text{Storeys}) + ((\sqrt[3]{\text{Nodes}} - 1) \times \sqrt[3]{\text{Nodes}} \times (\text{Storeys} - 1) \times 2)
\]

- Number of Restraint components. Upper limit:

\[
\text{Nodes} \times \text{Storeys}
\]
CHAPTER 3. IMPLEMENTATION

- Number of Node Mass components
- Number of Node Force components
- Number of Beam Distributed Load components

Also, there are three variants of this script, each altering the Nodes’ connectivity and serving a different purpose:

- Default: This variant does not assume full connection between Nodes. There can be partial or no connection between them at all (e.g. “rogue” Nodes, not connected to any other Node). This is the only variant in which the user can specify the number of Beams, Columns and Shells (Figure 3.1).

![Figure 3.1: 100 Nodes (red) and 50 Beams (green). No Node is not connected to every (or any) neighbor Node.](image)

- Fully connected: Beams: This variant assumes full connection between Nodes using Beams and Columns. Each Node is connected to its neighbors in the grid: up to three neighbors in the same storey using Beams, and up to two neighbors of the Storeys below and above using Columns. There are no shells in this variant. The number of Beams is fixed and exactly \(\text{Storeys} \times 2 \times \sqrt{\text{Nodes}} \times (\sqrt{\text{Nodes}} - 1)\). The number of Columns is fixed and exactly \(\text{Nodes} \times (\text{Storeys} - 1)\) (Figures 3.2a & 3.2b).

- Fully connected: Shells: This variant assumes full connection between Nodes using Shells. Each Node is connected to its neighbors (up to 5) in the grid with Shell components. There are no Beams, Columns, or Beam Distributed Load elements in this variant. The number of Shells is fixed and exactly \((\sqrt{\text{Nodes}} - 1)^2 \times \text{Storeys} + ((\sqrt{\text{Nodes}} - 1) \times \sqrt{\text{Nodes}} \times (\text{Storeys} - 1) \times 2)\) (Figures 3.3a & 3.3b).
3.3. INPUT FILES GENERATOR

(a) 100 Nodes, 90 Beams  
(b) 200 Nodes, 180 Beams, 100 Columns

Figure 3.2: Fully Connected Nodes with Beams and Columns.

(a) 100 Nodes (red), 81 Shells (dark blue)  
(b) 200 Nodes (red), 342 Shells (blue)

Figure 3.3: Fully Connected Nodes with Shells.

Each variant serves a different purpose. The first one (Default) aims to explore as many different combinations of number of elements as possible, even when the building is not fully connected. It also uses both analysis types (BEAM_ONLY & SHELL_ONLY) the solver offers. Nodes can be connected with both Beam and Shell components, which is the general case.

The second (Fully Connected: Beams) aims to explore fully connected configurations which maps to more realistic buildings/problems compared to the Default (if the Default is not fully connected). There are neither Shells nor “rogue” Nodes in this variant. BEAM_ONLY analysis type is used in this variant. Input files generated with this generator tend to grow really large in size compared to the other two (e.g. for the same number of Nodes), since Beam components have the largest XML description in terms of string size. As we will later see in 3.4, this leads to worse (string parsing and in-memory data structures assembling) / (parallelizable computation) ratios and less overall speedup for an increasing number of threads.

Finally, the third aims to explore fully connected configurations which maps to more realistic buildings/problems as well: There are no Beams, Columns, Beam Distributed Loads, or orphan Nodes in this variant. SHELL_ONLY analysis type is used in this variation. It turns out that input files made this way tend to have smaller size (e.g. for the same number of Nodes) compared to the other two, since
there are no Beam elements. This leads to larger (parallelizable computation) / (string parsing) ratios and lastly, greater speedup for the same number of threads assigned to them.

3.4 Profiling and benchmarking

One of the most important parts of this work was to discover as many parallelizable problems as possible, parallelization/high speedup problem configuration patterns, and finally, explore the problems’ variance in runtime and size as a function of their components. If there were not enough or even any high-speedup problems, or most/all of them were highly scalable, then this work would have much less point. In the first case, it would suffice to allocate just the bare minimum resources to each problem since they would not benefit from the extra ones. On the contrary, if most problems were highly scalable, it would suffice to allocate the optimal number of threads (w.r.t. the speedup gain) for all problems since they would benefit from them. Moreover, without the knowledge extracted from this part of the work we would not be able to properly create realistic data-sets for the predictor’s model training or the allocators’ evaluation.

However, as we will see in this section, this was not the case. Different problem components and different numbers of them lead to completely different execution times, file sizes and more importantly, speedup. As explained in 3.3, being able to massively create input files on demand was really helpful for this part of this work. We used the script described in the aforementioned section to automatically and massively create workloads.

Our main observations are the following: Firstly, components have different XML representations and size, thus, they contribute differently to the file’s size. Larger components’ parsing, in-memory allocation and data structures construction leads directly to less parallel/total runtime ratios. The number of components and file size relation is shown in Figure 3.4a. It is clear that Beams contribute linearly to an input file’s size. Other components do not seriously impact size and consequently do not increase serial runtimes.

Secondly, we can see in Figure 3.4b that the total runtime increases linearly with an increasing number of Shell and Beam components. The difference between the two however, is that Shells, due to their small XML representation and impact on file size, increase the “useful” computation times compared to Beams, which increase the XML parsing time. This will be seen clearly in later Figures again.

Each added component modifies the sparse matrix assembled (which the Cholesky algorithm factorizes). As shown in Figure 3.5, most components do not create (highly) parallelizable matrices except for Restraints. In this Figure, the building has the same (fixed) number of each component between experiments, but not simultaneously. It is clear that Restraints contribute the most to parallelization speedup, Thus, problems with adequate Restraint components can benefit from extra threads assigned to them.
3.4. PROFILING AND BENCHMARKING

(a) Components’ impact on file size

(b) Components’ impact on runtime

Figure 3.4: Components’ impact on file size and runtime

Next we explored the matrix assembling and Cholesky decomposition to total runtime relation which is shown in Figure 3.6. As we can see, the lack of Beam and Shell components (Node connectivity) gives the biggest assembling and decomposition percentage of runtime, up to 95%. On the contrary, problems fully connected with Beams have as low as 30%. The reason behind this is already explained: Beam components have much larger XML representation in terms of string size, which greatly grows parsing times. Such large parsing and allocation times dwarf the assembling and decomposition parts, which are the parallel parts.

In Figure 3.7a, we explored the maximum speedup possible for each problem variant. It is worth noting that all three problems consist of the same number of Nodes. Problems with Node and Restraint components achieve up to 77% speedup for 32 threads. Fully Connected: Shells (with Nodes, Shells and Restraints) up to 58%, while Fully Connected: Beams (with Nodes, Beams and Restraints) up to 32%.

As expected, the file size itself is not a good runtime and/or speedup indicator. This is not a surprise at this point but it was not obvious at all before this part of our work. As shown in Figure 3.7b, three different problems with a 15 MB XML file representation have totally different runtimes and speedup for the same number of threads. More specifically, problems with full Shell connections tend to have the largest runtimes (up to x20 compared to a Fully Connected: Beams problem of the same size), problems without Beams or Shells the greatest speedup (up to 77%) while Fully Connected: Beams the lowest runtime and the lowest speedup. This Figure does not contradict with the previous one. The reason why the FCB line is flat as opposed to the 32% speedup we saw in the previous figure is explained as follows: The first FCB building configuration has the maximum number of Restraints possible, and thus parallelization speedup, while the latter does not have any Restraints.

Finally, exploring the parallel part (burst) start and duration as a function
Figure 3.5: Component’s impact on speedup

of the building’s components could be of crucial importance for a few reasons: First and foremost, if a resource allocator could (almost) deterministically know a job’s burst duration then it could predict contention at burst time. What we mean by that is that the Solver’s (serial) initial parsing part of the runtime does not require more than on processor while the Cholesky (parallel) part may require more processors. At this (burst) time, there may not be any processors left or by allocating them to this specific job the next arriving jobs may have no resources to run on. Moreover, by accurately knowing a problem’s burst start and combining it with the accurate burst duration knowledge, we can achieve what we already mentioned: fine-grained CPU allocation. Lastly, if this analysis showed no correlation between components and/or configurations and burst time/duration, then we would need to find another method for fine-grained allocation.

Our results regarding the above are shown in Figures 3.8. $D$ refers to the Default variant, $DR$ to the Default only with Restraints, $FCB$ to Fully Connected: Beams, $FCBR$ to Fully Connected Beams only with Restraints, $FCS$ to Fully Connected Shells and $FCSR$ to Fully Connected Shells only with Restraints. As we can see, the simplest form of input files which is the Default with Restraint components only gives the highest percentage of burst duration relative to the total runtime (up to 50%). On the contrary, Fully Connected Beams give the lowest (even below 1%). Also, it is worth noting that as the number of threads increases the burst duration gets smaller, which is the parallelization’s effect on runtime. Finally, it is clear that there is indeed correlation between problem configuration
3.5 Data-set creation

After having a much better insight into the problems’ variance in runtime and speedup and by using the script described in 3.3, we created three sets of data:

- Data-set, 1,000 input files
- Test-set, 20% of the Data-set (200)
- Validation-set, 20% of the Data-set (200)

All sets were created in order to train, test, and validate the runtime predictor’s model (3.6). The reasons behind the data-set’s size choice are the following: Firstly, 1,000 files should be enough for a thorough search in the variables’ input domain as shown in Figures 3.9a, 3.9b, 3.10a, 3.10b. Secondly, in order to create the records and desired statistics we had to run the data-set multiple times. In order to determine if our results are noise-free, we had to run each file five times, take the average and calculate the standard deviation (of each desired statistic). Also, for each file we tried eight CPU limits (by setting the OMP_NUM_THREADS environment variable). Thus, we had $1,000 \times 5 \times 8 = 40,000$ runs. This should take a reasonable amount of time which holds in this case, while larger data-sets would take too long. Finally, 1,000 files (samples) should be enough for the model to
be properly trained, based on our observations of the previous section. This is equivalent to properly interpolate between the samples components’ values.

After the training “phase”, a model’s accuracy must be tested when the input is totally unknown, i.e. a problem it has never seen before. If applicable, the model’s hyper-parameter tuning takes place after the test phase. This tuning aims to push the model’s accuracy as high as possible. Again, in order to test the model’s new accuracy, we have to use a new (totally unknown) set of problems, the validation set. Thus, the validation set’s accuracy is the “fair” and correct one, if hyper-parameter tuning took place. The reason behind both the test-set’s and validation-set’s size choice is that 20% of the training/data-set is considered (empirically) the optimal size for accuracy checks, by people in the relevant field. All of the above sets consists of three file size “classes”:

- Small: $0 MB \leq \text{filesize} \leq 1 MB$
- Medium: $1 MB < \text{filesize} \leq 10 MB$
- Big: $10 MB < \text{filesize}$

All three sets consist of 50% small, 30% medium and 20% big, files. The reasoning behind this choice is the following: it is expected that at least half of the problems would be quite small in real-world situations, roughly $1/3$ of them medium sized, and only 15-20% of them large. However, the size threshold (1 & 10 MB) between the file size class splits is arbitrary, solely based on the size distribution of the initial (and small) “dataset” provided by the TOL staff. The file size frequency distribution is shown in Figure 3.11

Additionally, all three sets consist of the following problem “classes”:

- Normal: Problems created with the “default” generator, including all possible XML elements.
3.5. DATA-SET CREATION

Figure 3.8: Burst duration as percentage of the total runtime

- Normal: Restraints:
  Problems created with the “default” generator, including Restraints only. Thus only Storeys, Nodes and Restraints appear in this case.

- Fully connected: Beams:
  Problems created with the “Fully Connected: Beams” generator’s variant, including all possible XML elements (i.e. all minus Shells).

- Fully connected: Beams: Restraints only:
  Problems created with the “Fully Connected: Beams” generator’s variant, including Restraints only. Thus only Storeys, Nodes, Beams and Restraints appear in this case.

- Fully connected: Shells:
  Problems created with the “Fully Connected: Shells” generator’s variant, including all possible XML elements (i.e. all minus Beams, Columns and Beam Distributed Loads).

- Fully connected: Shells: Restraints only:
  Problems created with the “Fully Connected: Shells” generator’s variant, including Restraints only. Thus only Storeys, Nodes, Shells and Restraints appear in this case.

Again, the reasoning behind these choices, especially the “Restraints only” ones, is based on our benchmarking results described in the previous section. We wanted to cover all possible cases:
CHAPTER 3. IMPLEMENTATION

Figure 3.9: Nodes’/Beams’ values cover most of the input domain’s range

Figure 3.10: Shells’/Restrains’ values cover most of the input domain’s range

- **High speedup:**
  - Normal: Restraints
  - Fully connected: Shells (Restrains only)

- **Mediocre speedup:**
  - Fully connected: Shells
  - Fully connected: Beams (Restrains only)

- **Low speedup:**
  - Fully connected: Beams
  - Normal
3.6 Runtime prediction

A core component of our implementation is the runtime predictor, since both both (some) static allocators and the elastic depend on it. The runtime predictor is a regression model which takes a problem’s set of components plus a few variables as input, and gives the expected problem’s runtime, burst duration and burst start as output:

$$\text{Predictor} : U \rightarrow (\text{Runtime, Burst Duration, Burst Start})$$

$$U = \text{Storeys} \times \text{Nodes} \times \text{Beams} \times \text{Shells} \times \text{Restraints} \times \text{Forces} \times \text{Masses} \times \text{BeamDistributedLoads} \times \text{FileSize} \times \text{NumberOfThreads}$$  \hspace{1cm} (3.1)

In other words, the predictor is fed with the XML file’s components and size along with the number of threads the problem is going to be solved with, and gives the expected runtime, burst start and duration. Its main purpose is making accurate predictions for different number of threads for each file.

Example usage for a problem with 10 Storeys, 10,000 Nodes, 10,000 Restraints and of 5MB in size (burst start and duration are omitted):

```python
>>> regr.predict([[10, 10000, 0, 0, 10000, 0, 0, 0, 5000, 1]])[0]
48.60257621428575
>>> regr.predict([[10, 10000, 0, 0, 10000, 0, 0, 0, 5000, 4]])[0]
33.293502761904776
```

Figure 3.12: Different expected runtime for different number of threads (1 vs 4)

All predictor options we tried in this work are implemented in Python, using the scikit-learn [19] framework. From this point forwards and unless otherwise noted, all hyper-parameter-tuned regression model accuracy bars we show are based on the propel machine learning pipeline: Train, tune, validate, test. In addition, every configuration with 1.0 accuracy in the Figures does not reflect an 100% accurate
system. Instead, this is caused by the rounding (up) of the more than 6 digits the real numbers had.

### 3.6.1 Linear Regression Models

Linear regression was the first and obvious choice for a few reasons: Firstly, it is really simple to understand and use, probably the easiest of all training/regression models. Secondly, it seemed that the benchmarking process described in 3.4 showed clear correlation between certain elements and runtimes/speedup:

- Restraints with parallelization speedup.
- Shells with parallel/serial ratios
- Beams with lower parallel/serial ratios

Finally, it would be reasonable to expect linear relations between most Solver’s components and execution time.

#### 3.6.1.1 Simple Linear Regression

The simplest form of training and regression models, ordinary least squares Linear Regression. The training and testing results are shown in Figure 3.13. As we can see, neither fit_intercept nor normalize configuration options’ combinations gave more than 80% test-set accuracy.

![Figure 3.13: Linear Regression with fit_intercept and normalize options](image)

(a) Train accuracy  (b) Test accuracy

#### 3.6.1.2 Ridge

Since the “simple” linear regression didn’t give adequate accuracy, we tried linear least squares with a regularization factor (Ridge regression or Tikhonov regularization). Ridge regression is used for data that suffer from multi-collinearity. This
3.6. RUNTIME PREDICTION

model adds a degree of bias to address this issue. The training and testing results are shown in figure 3.14. We tried a few different configurations of Alpha (the regularization strength) and internal Solvers (Singular Value Decomposition, Cholesky, Stochastic Average Gradient descent etc.), but none gave more than 80% accuracy.

![Figure 3.14: Ridge Regression Accuracy](image)

3.6.1.3 LASSO

Since neither “simple” linear nor Ridge regression gave adequate accuracy, we tried a Linear Model which applies a regularization as well (LASSO). This regularization is usually applied to problems with a few features. Some features (their coefficients) can become zero and be eliminated from the model. Larger penalties result in coefficient values closer to zero, which is the ideal for producing simpler models. The training and testing results are shown in Figure 3.15. We tried a few different configurations of Alpha (regularization strength), tolerance for the optimization and maximum number of iterations it performs, but no one gave more than 80% accuracy as well.

![Figure 3.15: LASSO Regression Accuracy](image)
3.6.1.4 ElasticNet

The last linear regression resort was to try linear regression with the combined regularizers of LASSO and Ridge methods. We thought that the combined effects could yield better accuracy. The training and testing results are shown in Figure 3.16. We tried a few different configurations of Alpha and the relative regularizer ratio (which one is “stronger”), but no one gave more than 80% accuracy.

![ElasticNet Regression Accuracy](image1)

### Figure 3.16: ElasticNet Regression Accuracy

3.6.2 Polynomial Regression

At this point it was clear to us that we were dealing with non-linear relations between certain components and runtime. For this reason we tried polynomial regression with degree up to 4. The training and testing results are shown in Figure 3.17. We achieved more than 90% accuracy, for the first time, with polynomial regression. At this point we wanted to push it a bit more by trying a couple of methods and seeing if we can do better.

![Polynomial Regression Accuracy](image2)

### Figure 3.17: Polynomial Regression Accuracy
3.6. RUNTIME PREDICTION

3.6.3 Support Vector Machine Regression

Support Vector Machine (SVM) regressors support a lot of hyper parameters to tune. We tried various configurations of C (regularization parameter), kernel type to be used in the algorithm (linear, polynomial, sigmoid) and Epsilon (a threshold after which a penalty is issued on a coefficient). SVMs can achieve up to 95% accuracy with proper hyper-parameter tuning. The training and testing results are shown in Figure 3.18.

![SVM Regression train accuracy](chart1)

(a) Train accuracy

![SVM Regression test accuracy](chart2)

(b) Test accuracy

Figure 3.18: SVM Regression Accuracy

3.6.4 Random forest Regression

Finally, due to the SVMs quadratic fit time complexity which led to huge runtime for every training, we decided to try a random forest regressor (RFR). The training and testing results are shown in Figure 3.19. As we can see, RFRs can achieve up to 98% accuracy even without hyper-parameter tuning. Considering that they are really fast as well, it was our final choice for our elastic allocator’s runtime predictor.

![Random Forest Regression train accuracy](chart3)

(a) Train accuracy

![Random Forest Regression test accuracy](chart4)

(b) Test accuracy

Figure 3.19: Random Forest Regression Accuracy
3.7 Baseline Resource Allocators

One possible rival argument against our work could be that with only 10% “big” problems and/or only a few highly parallelizable problem types in real-world scenarios, dynamic/elastic resource allocation is needlessly complex and an overkill for this application. In order to show that even with such rarity the overall gains outweigh the implementation and/or runtime cost, we designed and benchmarked simple resource allocation strategies which we will evaluate and compare with the elastic one. All policies presented in this section are static, meaning that the assigned resources are allocated at job creation time and do not (dynamically) change during the job’s runtime.

The allocators follow the design shown in Figure 3.20. Dotted arrows indicate file reading/writing. Ellipses indicate description files: The problem’s XML description (beige/cream) and the pod’s YAML description (blue). Rectangles indicate scripts: The red one is the resource allocation bash shell script. The green one is the Python runtime predictor we discussed in previous sections. Finally, the rounded corners rectangles indicate executables: The allocator’s script calls the Kubernetes runtime in order to create and start the pod which run the Solver.

The first step is to give the XML file’s name (0). The allocation script reads the file and stores the components’ value (1). It then calls the runtime predictor with these values as arguments and gets the runtime prediction as a result (2 & 3). Based on the allocation policy, it creates a YAML deployment file with the appropriate resource limits for each problem (4). Finally, it creates the pod by calling `kubectl create -f filename`, where `filename` is the unique YAML deployment file (5). There are a few things to note: Firstly, the XML file is stored in an NFS directory which is mounted on each pod at creation time. This way the Solver’s container can read the XML file and solve the problem it describes. Secondly, the vCPU limits are set with the Kubernetes’ `request` and `limit` feature (5.1). Lastly, the simplest two allocation policies do not utilize the runtime predictor (steps 2 & 3 are omitted): The simple, static resource allocation which allocates a fixed number of vCPUs for every problem independently of any variable (3.7.1) and the static resource allocation which allocates a fixed number of vCPUs based solely on file size (3.7.2).

The YAML deployment files’ template is shown in Figure 3.21. The name field is the job’s name which has to be unique, because Kubernetes cannot create jobs with same names. The CPU request and limit will be set to an equal value based on the allocation policy’s decision. Finally, the Solver’s argument is the XML file name to load and read from the NFS mounted (by the pod) directory.

3.7.1 Static resource allocation

The simplest possible allocator, which statically assigns the same resources (vCPUs) for every problem/job. Such an allocator is trivial to both understand and
3.7. BASELINE RESOURCE ALLOCATORS

3.7.2 Static resource allocation based on file size

This policy statically allocates different number of vCPUS for each job, based solely on file size. Without the extensive benchmarking and parallelization exploration we did, it could be easily assumed that larger files translate to larger runtimes (which is not entirely accurate) and worse, larger speedup gains. Thus, it would make sense to allocate vCPUS as a function of problem size. As we shown in
previous sections however, this is not the entire picture and allocating resources based on how big a problem seems to be would be a naive (but possible) approach. The allocation function is defined as follows:

\[
\text{Resource allocation}(\text{file size}) : \begin{cases} 
\text{small} & \text{if } 0 < \text{file size} \leq \text{threshold} \\
\text{medium} & \text{if } \text{threshold} < \text{file size} \leq 10 \times \text{threshold} \\
\text{large} & \text{if } 10 \times \text{threshold} < \text{file size} < \infty
\end{cases}
\]

We explored the following possible values for each variable:

\[
\text{threshold} = \{500, 1000, 3000, 5000\} \text{ KB} \\
\text{small} : \{1, 2, 4\}, \text{ medium} : \{4, 6, 8\}, \text{ large} : \{8, 12, 16, 32\}, \text{ vCPUs}
\]

\[
\text{Total configurations} = |\text{threshold}| \times |\text{small}| \times |\text{medium}| \times |\text{large}| = 144
\]

### 3.7.3 Static resource allocation based on expected runtime

This policy statically allocates vCPUs as a function of expected runtime. More specifically, we use the runtime predictor to guess the problem’s execution time and

```yaml
---
apiVersion: batch/v1
kind: Job
metadata:
  name: # TO BE FILLED
spec:
  template:
    metadata:
      name: raf-solver
      labels:
    spec:
      restartPolicy: Never
      containers:
        - name: raf-solver
          image: rafsolver:latest
          resources:
            requests:
              cpu: # TO BE FILLED
            limits:
              cpu: # TO BE FILLED, SAME WITH REQUEST
          command: ['/bin/Solver*']
          args: ['"/usr/src/files/'] # TO BE FILLED
          imagePullPolicy: IfNotPresent
          volumeMounts:
            - mountPath: /usr/src/files
              name: files
---
```
assign more resources as the execution time increases. This is a better approach since the expected runtime is a slightly more accurate indicator of a problem’s scalability. Moreover, larger runtimes yield more absolute-time gain even when marginally sped-up, compared to shorter runtimes. For example 15% speedup on a problem with 1000 seconds expected runtime is 150 seconds gain, while a 50% speedup on a problem with 20 second expected runtime is “only” 10 seconds less execution time. The main issue with this approach is that there are a lot of problem configurations that have extremely large runtimes but most of it is simple XML parsing and serial matrix population, not parallel-izable work.

The allocation function is defined as follows:

\[ vCPUs_{\text{allocation}}(\text{expected\_runtime}) = \begin{cases} 1 & \text{if } \text{expected\_runtime} \leq \text{threshold} \\ \text{threads} & \text{if } \text{expected\_runtime} > \text{threshold} \end{cases} \]

We explored the following possible values for each variable:

- \( \text{threshold} : \{5, 10, 20, 50, 100\} \) Seconds
- \( \text{threads} : \{2, 4, 6, 8, 12, 16, 32\} \), vCPUs

Total configurations = |threshold| \times |threads| = 35

### 3.7.4 Static resource allocation based on expected speedup

Finally, this resource allocator calculates the expected speedup for a specific number of threads and if it is above a given threshold, it assigns this number to the job/pod. There are many possible configurations for this policy since there could be multiple speedup “steps” and thresholds, accordingly. For example, one extra vCPU should yield at least 10% speedup in order to be “worth it”, two extra 25% and so on so forth. This is the best policy so far mainly because it is the only one accounting for true/raw speedup for the assigned resources. The only (and final) issue that this approach faces, along with all previous ones of course, is that it does not account for changes in the incoming traffic (e.g. flash crowds). The proposed elastic allocator combats this and similar weaknesses and successfully overcomes them, as we will later see in the evaluation section (4).

The allocation function is defined as follows:

\[ vCPUs_{\text{allocation}}(\text{expected\_speedup}) = \begin{cases} 1 & \text{if } \text{expected\_speedup} \leq \text{threshold} \\ \text{threads} & \text{if } \text{expected\_speedup} > \text{threshold} \end{cases} \]

We explored the following possible values for each variable:

- \( \text{threshold} : \{10, 20, 30, 40, 50\} \) Percent
- \( \text{threads} : \{2, 4, 6, 8, 12, 16, 32\} \), vCPUs

Total configurations = |threshold| \times |threads| = 35
3.8 Vertically Elastic Resource Allocator

An elastic resource allocation policy should avoid all scaling issues mentioned in the previous sections:

- Variable resource demands as a function of the problems’ configuration: Different component configurations have different scalability margins, hence resource demands.
- Dynamic resource demands as a function of time: A job’s demands change between serial and parallel parts of its execution time.
- Excess allocation: Redundant resources, under-utilization, waste of money.
- Under-allocation: Narrow speedup margins, less parallelization exploitation, larger overall execution times.
- Variable traffic rate: When increasing, SLO violations occur due to higher waiting times. When decreasing, under-utilization and resource waste can occur.

Our proposed elastic allocator successfully resolves the aforementioned issues in the following ways: The allocator’s predictor component can accurately predict each problem’s possible speedup for a specific number of threads/vCPUs assigned to it. Additionally, the predictor can also predict at which point of its execution time and for how much a problem could utilize extra threads/vCPUs. Excess/under allocation is avoided since the allocator “knows” exactly how many resources and when to assign to each problem. Lastly, the allocator elastically adapts to traffic rate changes by carefully and dynamically allocating vCPUs based on the average incoming rate, cluster’s load, and problems’ possible speedup. Our assumptions and simplifications are the following:

- The proposed algorithm supports virtual CPUs only as a resource although it can be easily extended to other resources such as memory, storage and network.
- We assume a single node setup where pod migration and/or node management is not needed. Without node management and horizontal/cluster scaling we can have a simplified model easy to both understand and implement.
- We have a vCPU limit of 32 mainly for two reasons: The Solver’s parallelization rarely gains speedup for more vCPUs or even loses after 16. Also, our evaluation machine has 32 physical threads and we needed to avoid over-subscribing.

We present the elastic vCPU scheduling and allocation pseudo-codes (Algorithms 1 & 2):
Algorithm 1 schedule()

1: window $\leftarrow [x]$ //last x arrival times, configurable
2: counter $\leftarrow 0$
3: last_arrival_time $\leftarrow 0$
4: while (1) do
5:     while no_network_or_alarm_interrupt() do
6:         {;
7:     if (network) then
8:         time $\leftarrow$ system.getTime()
9:         window[counter mod x] $\leftarrow$ time – last_arrival_time
10:        last_arrival_time $\leftarrow$ time
11:        file $\leftarrow$ load_file_from_nfs()
12:        burst_start $\leftarrow$ get_burst_start(file)
13:        pid $\leftarrow$ start_pod(file)
14:        alarm(burst_start, pid)
15:        counter $\leftarrow$ counter + 1
16:     else if (alarm, pid) then
17:         allocate(pid, file, window.mean())
18:     

Algorithm 2 allocate(pid, file, mean_arrival_time)

1: cpu_limit $\leftarrow$ system.getIdleCPUs()
2: if cpu_limit < 2 then
3:     exit();
4: choices $\leftarrow \{2, 4, 6, 8, 12, 16, 32\}$
5: serial-runtime $\leftarrow$ runtime_predictor(file, 1)
6: score $\leftarrow 0$
7: vCPUs $\leftarrow 1$
8: for i in choices and i $\leq$ cpu_limit do
9:    parallel_runtime $\leftarrow$ runtime_predictor(file, i)
10:   burst_duration $\leftarrow$ get_burst_duration(file, i)
11:   if burst_duration $>$ mean_arrival_time then
12:       continue;
13:   speedup $\leftarrow$ (1 – (parallel_runtime/serial_runtime)) * 100
14:   new_score $\leftarrow$ speedup/i
15:   if new_score $>$ score then
16:      score $\leftarrow$ new_score
17: vCPUs $\leftarrow i$
18: if vCPUs $>$ 1 then
19:   change pod’s (with pid) cgroups CPU limits
20: return
CHAPTER 3. IMPLEMENTATION

Schedule: The allocator’s elasticity should be, by definition, elastic to traffic rate changes. For this reason the allocation algorithm accounts for the average problem arrival rate. The last $x$ problems’ arrival times are stored in an array accessed in a round-robin fashion, where $x$ is configurable (line 1). Then, at burst time, the average arrival time is passed as argument to the allocation function (line 17). Lines 5-6 abstract the blocking nature of the service which would serve the requests. Wait as long as no events happen. If however, a network interrupt occurs (meaning a problem to solve arrived), we then execute the code in lines 7-15. More specifically, we calculate the difference between the arrived problem’s time and the previous one and store it in the array (lines 8-10). Next, we load and read the file stored in a specific directory in the NFS (line 11). At this point we can predict the problem’s burst start time based on its components’ values (line 12). Finally, we can start the pod and set an alarm interrupt which will enable us to call the allocator function just before the pod’s burst start (lines 13-14).

Allocate: We firstly need to know how many CPUs are idle/available (line 1). If there are less than two, we simply cannot scale the pod’s vCPUs (lines 2-3). Then, we need to predict the problem’s expected serial runtime (line 5) in order to calculate the speedup with the new number of vCPUs (line 13). Then, we try different vCPU allocation options in order to determine if and which is the optimal one (lines 8-17). Firstly, the parallelized runtime must be predicted (line 9). The burst duration can be predicted as well (line 10) and if it is larger than the mean arrival time, we skip this option (lines 11-12). The reasoning behind this choice is that we want to eliminate/minimize waiting times and SLO violations. In the case of long enough burst durations, the next arriving problems may wait for resources to be freed, and starve. Also, burst durations tend to decrease as the number of threads increases which is desired for both higher speedup gains and lower chance of having non-zero waiting times. Finally, we calculate a score which essentially is the speedup gain over the resource “cost” we pay for that gain. If the new score is greater than what we currently have, we choose this number of vCPUs to scale to (lines 14-19).

We argue that our proposed algorithm is as fine-grained as possible for the following reasons: First of all, we allocate vCPUs at a granularity of multiples or powers of two. This is the optimal choice assuming that each pair of threads assigned to a process will run on the same physical core, sharing low level caches and having lower communication delays. Also, both hardware and OS are aware of (and optimized for) this property. Therefore it would be pointless or even sub-optimal to allocate odd number of vCPUs. Moreover, from our profiling we know that the Solver cannot gain (non-marginal) speedup with more than 32 vCPUs. Additionally, we allocate the aforementioned vCPUs exactly at burst time, when they are needed. This translates to almost perfect utilization according to our definition of utilization. Almost, because our profiling analysis also showed that each additional pair of vCPUs runs for less time compared to the previous. For example, a job running with 8 threads and of 60 seconds total runtime could have the following thread lifetimes: 1st: 60 seconds, 2nd 20 seconds, 3rd and 4th:
18 seconds, 5th and 6th: 14 seconds, 7th and 8th: 10 seconds. We would allocate 7 extra threads for a burst duration of 20 seconds but not all threads will be actually running for that time. This is not a problem, however, since granularity at even finer grains is almost impossible and totally pointless (we would need vCPUs/2 alarms in order to timely reduce the cgroups CPU limits to the “optimal” level). At this point, we should clarify that the reason we do not need to reset the cgroups limits after the parallelization spike is because the Solver completes almost immediately after this part. Thus, it would be needlessly costly to once again interrupt the scheduler and do the cgroups change, for a mere 1-2 seconds (worst case) of runtime left.

This algorithm can be optimized in various ways. Our first thought is to transfer the speedup computation to the schedule function. If there is no speedup or it is marginal (relative to a configurable threshold), then never set the alarm up and consequently call the allocator function. The schedule function predicts the expected burst start regardless. Predicting the speedup gain for two (or any number) vCPUs allocation would incur a smaller overhead than the interrupt and allocate function call which would exit halfway in this case. An additional optimization would be to schedule jobs in a shortest job first fashion. This means that in the case of a job with high expected runtime (again, above a configurable threshold), it may be more optimal to wait for the next job, especially in high traffic rates. If this (and any upcoming) job(s) have a substantially smaller runtime, schedule them. This translates to lower waiting times for them and better overall quality of experience for the respective users. The third optimization that could yield performance would be to store the currently running pod’s burst durations. If, at allocation time, there were not (enough) idle vCPUs, it might be worth to wait for some vCPUs to be freed. Given a small burst duration, the slightly higher waiting time (increased by the pod suspension) may be worth paying for a (much) shorter execution time, due to parallelization speedup. This optimization greatly increases the implementation complexity, however. Finally, even if a problem has a burst duration larger than the average arrival rate, it may be still be optimal to allocate a few additional vCPUs to it. In order to optimally do this, one has to consider the following: the burst duration to average arrival time ration, the number of additional vCPUs, the speedup gain from those vCPUs. Again, this increases the overall system complexity and needs careful consideration.
Chapter 4

Evaluation

Our benchmarks and profiling described in 3.4 were executed on a 2.0 GHz AMD EPYC 7551P 32-Core Processor with a total of 16 hyper-threaded cores (32 hardware threads), running on CentOS 7 and kernel version 3.10.0. Also, all baseline allocators were benchmarked and evaluated on the exact same setup, in order to select the best one of each variant. The best from each variant is compared to the elastic, in this chapter. The elastic allocator’s runtime and traffic, however, are not implemented and evaluated on an existing setup, but simulated, for the following reasons:

- The current Kubernetes version does not support live (dynamic) resource limits modification without restarting the pod. This is destructive for the elastic allocator since it depends on adding resources at the exact moment the pod needs them, without evicting and restarting it.

- Even if the above feature was supported, it would be not trivial at all to add a custom scheduler and allocator as a plugin. There are multiple Kubernetes components that need customization and/or modifications. We estimate that it would need at least another thesis like this, in time and effort.

- Without using Kubernetes, for example by relying on a single-node setup, it would take less effort to actually deploy the allocation itself. This would either disable node scaling or force a custom implementation of it from scratch, as a trade-off. Manually changing the cgroups limits during the runtime as [1] did, and book-keeping the appropriate information regarding each docker container running, when it would need the excess resources, etc, would be doable in the scope of an undergraduate thesis. Still, not fast enough for this work to be presented on time.

The simulation process was implemented as follows: The hyper-threaded cores are simulated as counters. Zero means the thread is idle while a positive number (\(\text{count}\)) indicates that this thread has \(\text{count}\) seconds of work still left. Each time a problem “arrives”, the problem’s runtime, average burst duration, maximum burst
duration, and burst start are predicted with help of the runtime predictor. Then, the CPU counters are decremented by the problem’s “arrival” time. This way, we simulate the passage of time between problem arrivals. The counters are sorted for the purpose of simulating the first available CPU, then the next available and so on so forth. After that point, the first (and maybe last) vCPU allocated for the specific problem is simulated by simply adding the problem’s predicted runtime to the first counter (the one with the smallest value, since it would be the first available/idle). With subsequent problem “arrivals”, the passed time is calculated. If it is equal or more than the problem’s burst start time, the allocation algorithm decides the extra vCPUs, if any, to assign to this problem, as described in 3.8. The maximum burst duration is simply added to the sorted counters (excluding the one the problem “runs” on). This way, all vCPUs (finally) allocated to this problem accurately reflect their total busy time.

Waiting times are calculated as follows: every time a problem has no available cores to run on (there are no zero counters), the difference between the arrival time and the smallest counter’s value is the problem’s waiting time.

Utilization is calculated as follows: the ideal vCPUs execution time is the problem’s runtime multiplied by the number of vCPUs. The real vCPUs execution time is the problem’s runtime (one vCPU with 100% utilization) plus the average burst duration multiplied by the number of vCPUs-1. Then, utilization is \( \frac{\text{real execution time}}{\text{ideal execution time}} \times 100\% \). At this point it should be clear why utilization is not always 100\%: \( \text{average burst duration} \leq \text{max burst duration} \). Additionally, we can neither predict nor allocate vCPUs at perfect granularity, i.e. for every vCPU’s the exact busy time. Thus, it is only possible to allocate for the maximum (between vCPUs) burst duration which of course leads to some under-utilization.

For the remainder of this chapter and unless otherwise noted, the following holds: Static Size has a 1.000 KB threshold between small and medium and 10.000 KB between medium and large problem sizes. Also, small is configured as 1 vCPU, medium as 4 and large as 8. Static Runtime is configured with a threshold of 20 seconds and threads of 6 vCPUs. Finally, Static Speedup is configured with a threshold of 20% speedup and threads of 6 vCPUs as well. Each point in the upcoming figures is the average of 5 runs (applies to the real runs and not the simulations). The problem arrivals are simulated as follows: First, a random permutation of the dataset is created. Then, for a subset of this permutation, all five implementations (4 static + 1 elastic) are evaluated. Lastly, repeat this process 10 times. Therefore, each “average” metric in the upcoming figures is the average of the 10 random permutations of jobs. Apart from that, it obviously is the mean time (of the according metric) between jobs. E.g. The average completion time is the sum of the jobs’ completion times divided by the number of jobs. The average completion time figure shows the average of 10 such 10 averages.

Next we show evaluation Figures for the aforementioned setup and simulation: In Figure 4.1 the average problem/job completion time is shown. The x axis is the rate of problem arrival/traffic as problems/minute. The y axis is the average
job completion time in seconds. We can notice that in most static approaches the completion time increases (almost) linearly with traffic rate increase. This is a common lack of elasticity implication. As we can see, our elastic approach (purple line) has the lowest average from 3 to 8 jobs/minute. With less than three jobs per minute the other approaches have less completion times due to the (over)generous allocations they issue, thus parallelization speed. After 8 jobs per minute the simplest static approach marginally (1.2%) surpasses ours because of the strict (and trivial) allocations it issues. In addition, the reason behind the slightly higher completion times at high traffic rates is due to some parallelism added compared to the static, single vCPU approach. In total, during low traffic rates the elastic approach gives up to 20% less average completion time compared to the simplest static one. Also, it is only up to 7% slower compared to the lowest averages of the static size approach. During high traffic rates our approach leads to up to 15% lower job completion times compared to the static size one.

![1 Node, 32 vCPUs: Average completion time](image)

Figure 4.1: Baseline vs elastic allocation: Average job completion time

In Figure 4.2a we can see the average problem/job execution time on y axis, in seconds, and the rate of problem arrival on x axis, in problems/minute. All static approaches have a flat execution time because no matter what the arrival rate is, the problems always have the same number of resources assigned to them, thus same parallelization exploitation (if any). The elastic approach however, can easily adapt to traffic changes. At first, during relatively small arrival rates, the execution times are quite low due to the abundant vCPUs allocated to jobs, hence parallelization speedups emerge. As the arrival rate increases the execution
time increases as well since allocation is stricter due to finite resources limitations and waiting times minimization. Average and maximum waiting times must be kept at zero or as low as possible. The simplest static approach has the highest execution time due to fixed and only one vCPU assigned to each problem, hence no parallelization at all. On the contrary, the static approach based on file size has the lowest execution times due to generous vCPUs allocations (quite a lot of problems are bigger than the thresholds mentioned earlier in this section).

In Figure 4.2b, the average waiting time is shown on y axis while problems/minute is on the x axis. It was expected that the static single vCPU approach would have the lowest waiting times since it has the most stingy resource allocation and thus, contention, possible. Indeed we can see that this is clearly the case. The elastic approach follows immediately which adapts quite well to traffic changes. The rest approaches have an almost linear relationship with the increase in traffic. However, average waiting times do not provide really helpful information in the cloud domain. To this end, we calculated/predicted maximum waiting times and tail latency as well.

![Average execution time](image1)
![Average waiting time](image2)

Figure 4.2: Average completion time breakdown: Execution time (left) & waiting time (right)

Figures 4.3a & 4.3b show the maximum waiting time and percentage of waiting time larger than 60 seconds, respectively. First and foremost, both elastic and single vCPU allocation strategies achieve a maximum waiting time of just below or equal to 60 seconds, which we consider to be the SLO violation threshold. On the contrary, the other three methods have linearly increasing maximum waiting times for once again (w.r.t. the increasing traffic rates). Similar results are shown in Figure 4.3b. 100% of the problems are solved with less than 60 seconds waiting time in the case of single vCPU assignment. Next, our approach follows with 5.3% at worst case (maximum “realistic” arrival rate). The rest approaches diverge really fast and lead to up to almost half of the problems to have delays more than 60 seconds. This means that our method achieves up to x10 less SLO violations compared to the static size method and up to x6 compared to the static speedup.
method. Therefore we provide users with higher quality of experience compared to simpler approaches.

![Graph](image1.png)

(a) Maximum waiting time  

![Graph](image2.png)

(b) % of waiting times > 60 seconds

Figure 4.3: Maximum waiting times (left), Waiting times > 60s percentages (right)

Then, we can see in Figure 4.4 that the elastic allocator achieves more than 88% utilization and up to 95%. The reason all other lines are flat is because of the way utilization is defined, which is traffic independent. The static size approach is by far the worst since it abundantly allocates vCPUs to a big portion of the incoming problems meanwhile most of them are non parallelizable or have really low speedup margins. The static single vCPU approach obviously utilizes 100% of the cores allocated for each problem (i.e. 1). Overall, the static speedup policy utilizes 12% less its resources while our policy only 7% on average compared to the single vCPU one.

The next and final Figure, 4.5, shows the allocators’ scaling as the number of vCPUs increases. Completion times get lower as the number of vCPUs increases mainly due to less contention and therefore lower waiting times, not scalability per se. Our policy follows the single vCPU assignment policy closely until 24 vCPUs. Then, it outperforms it by a margin of 8% along with all the other policies. In configurations with less vCPU (8), our approach gives up to 12% less overall job completion times.

In conclusion, we can safely argue that a fine-grained vertically elastic vCPU allocation policy can achieve less SLO violations, higher allocated vCPU utilization and better overall quality of experience, compared to simpler static policies. Note that even the “simple” scheme based on the expected speedup is already pretty good for application that follow speedup patterns similar to the Solver’s. Without extensive profiling, benchmarking and predictor models, even this scheme would not be available for deployment.
Figure 4.4: Baseline vs elastic allocation: Average waiting time

Figure 4.5: Baseline vs elastic allocation: Average job completion time
Chapter 5

Related Work

5.1 Kubernetes’ resource request and limit

Kubernetes supplies developers with elegant ways to specify their applications’ resource needs. Such one is the (CPU) request and limit specification which can be applied to pods and more specifically to the containers contained within [17]. The developer can set the container’s minimum resource requirements by setting a request for the according resource. In our case one vCPU would be the bare minimum computing requirement. Similarly, they can specify the maximum requirements by setting a limit for the according resource. Kubernetes then guarantees that those minimum resources are always allocated (by scheduling the pod on a node with enough resources available) and that the limits are never exceeded. It is worth noting that the scheduler will not schedule a pod if the sum of the requested resources of the running pods and the to-be-scheduled pod would exceed the node’s capacity, even if there is no actual usage of those resources.

In our case, by properly setting the request and limit for each arrived problem/job, under/over utilization of the allocated resources could be avoided to some extent. For example, with the help of the runtime predictor one could predict the runtime for a single vCPU allocation, then for various numbers of vCPUS, calculate the speedup (if any), and assign a vCPU limit greater than one only if the speedup was above a threshold. This way it would be guaranteed that every job would have at least one vCPU allocated to run on, but if more resources were available they would be assigned to it job as well. This is an improvement over the static speedup baseline allocator we described in previous chapters. The static speed allocator can only assign a fixed number of vCPUs which is greater than one even in the cases of contention. Having the ability to allocate just one vCPU at pod creation time and then scale it if needed would be a major improvement.

This approach, however, suffers from the following issues: Firstly, dynamically changing a running pod’s cgroups limits is not supported yet. After such modification the pod will be evicted and restarted. This usually negates or greatly reduces any potential speedup gains due to the potentially large
serial part at the start. Only a few, very specific problems have large enough runtimes and speedup to still benefit from a complete eviction and restart with extra resources.

The second problem is that there is neither best speedup threshold nor best number of vCPUS for all cases. For small thresholds (e.g. 5-20%), a flash-crowd can easily exhaust all available resources because most pods will have more vCPUS than their request assigned to them, until there are non left. The following jobs will have to wait for resources to be freed in order to be scheduled. This will greatly reduce QoS since total completion times will be high due to larger waiting times. For large thresholds (e.g. 40+%) and low incoming traffic rates, QoS will be reduced as well since total completion times will be quite high due to larger execution times, despite lacking waiting times. There would be unallocated and idle resources which could be properly be utilized.

5.2 Kubernetes’ vertical pod autoscaler

The second vertical scaling feature is Kubernetes’ Vertical Pod Autoscaler (VPA) [16]. VPA automatically and dynamically scales resources at runtime based on the analysis of historical data and resource availability in the cluster. VPA’s core component is the recommender which as its name suggests, recommends specific resource size allocation at both pod creation and burst time. Then, during the pod’s runtime the updater updates the pod’s limits to match with the recommender’s suggestions for the resource demands spike. VPA lacks configurable and custom recommendations, although the users can specify an upper and lower bound to the recommendation algorithm thus limiting its possible range of resources allocation. The recommender watches the running pods in the cluster and co-factors their historical resource utilization in order to determine its recommendation.

There are three main issues with this scheme. when applied to our target scenarios of course. Firstly, VPA evicts and restarts the pod in order to update the newly recommended resource limits. As we stated earlier, this can be disastrous in our case since the speedup gain from the extra resources may be less than the (double) initial XML parsing runtime cost. Second, it is harder for pods with very spiky resource usage patterns to get the sweet spot recommendation between balance and performance, compared to flat-lined resource demands pods. VPA heavily depends on historical data which may not have any relation to the current (to be scaled) pod. Consider the following example: The first 10 Solver’s problems to run on the cluster do not have any parallelization and thus, CPU demands. Then, a heavily parallelizable problem arrives. Most probably, the VPA’s recommender will decide that the according pod will have no resource requests. On the other hand, the recommender can easily grant excess resources which will not be properly utilized due to the problem’s non highly parallelizable configuration. Last but not least, it is not clear and transparent to the user how the recommender analyses the historical data for its recommendation.
5.3 Other work in academic literature

PRESS [13] is an online resource demand prediction system for elastic cloud scaling. It offers two prediction modes: signature-driver and state-driven. The former uses signal processing methods (such as Fast Fourier Transform) in order to decide if and what repeating demand patterns exist. The latter depends on collected metrics/logs, based on which it tries to predict the resource demands in the immediate future. Firstly, PRESS targets Virtual Machines and not Containers, which tend to behave differently due to their lightweight-ness. Secondly, this system is application-agnostic, in contrast to what we try to be. We argue that neither of the above methods match our scenario (exactly) for the following reasons: Job-dependent parallelizable problems rarely have repeating patterns except for the case the application’s users have. In our case, that is certainly not the case. On the other hand, metrics and near-future prediction would not work perfectly, being problem-oblivious. Our approach, and more specifically, prediction, almost perfectly “knows” each problem’s ideal demands.

CloudScale [20] builds upon PRESS, making it a prediction-based elastic resource scaling system as well. It tries to correct the underlying prediction’s error by adding prediction offsets, which aim to avoid under-allocation, and reactive error correction. In addition, it adds the job migration feature in order to predict possible resource exhaustion and the subsequent job migration on a less loaded host. For the first part, we argue that CloudScale has the same limitations that PRESS has. It targets VMs, is application-oblivious and can not (almost) perfectly predict an application’s spiky CPU demands, which depend solely on the incoming problem configuration. For the second part, we already stated that our allocator does not handle pod migration for now, but based on this and similar work could be implemented in the future.

ElasticDocker [1] is one the first and most important works on container vertical elasticity. It depends on live monitoring systems and resource-adjusting controllers. The elastic allocation algorithm can be summarized in three steps: monitor, analyze, scale. The elasticity controller scales the container’s resource limits based on the metrics analysis’ decision and arbitrary thresholds, e.g. less than 70% CPU utilization leads to down-scaling while more than 90% to up-scaling. Similarly to CloudScale, ElasticDocker supports container live migration for overloaded host off-loading and better container scaling. This system faces two of the three issues the previous systems had: Application’s behaviour ignorance and imperfect prediction. For example, the RAF::Solver will certainly exceed the 90% utilization cap for its one, initially allocated vCPU. However, the second vCPU to be allocated may be greatly under-utilized which could be avoided by simply predicting the applications true needs.

The above list is not exhaustive but rather indicative, reflecting our observations of the most influential and “impact-eous” work on vertically elastic resource scaling for the cloud. Our methodology may help others build fine-tuned, custom-tailored schedulers and allocators for their (very) specific needs.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work we ported to Linux, Dockerized and deployed as a cloud service a Greek structural design application. We carefully designed an input files generator and then generated data-sets on scale. Each contained file reflects realistic building scenarios. We thoroughly studied the Solver’s runtime behaviour such as the input to runtime and speedup relation, XML parsing time to Cholesky solving time ratios and finally, the burst start and duration respective to the total runtime. This analysis motivated us to deploy supervised learning methods and train regression models which accurately predict a problem’s runtime behaviour. Based on this analysis we built a fine-grained vertically elastic virtual CPU allocator. Our proposed allocator targets scenarios that match the above characteristics and co-factors varying incoming traffic rates. Our evaluation study shows that vertically elastic policies custom-tailored to specific resource demands can achieve up to ten times less SLO violations, 77% higher resource utilization and better overall quality of user experience.

6.2 Future work

First and foremost, we plan to actually deploy our proposed allocation scheme and then evaluate it on a real system. By doing this we can prove in a more concrete way that our runtime/traffic simulation was as realistic as possible and the presented gains match the reality. After that point we can properly optimize our allocation scheme since the performance bottlenecks will be more clear to us. Another possible step towards more realistic evaluation would be to compare our proposed scheme with Kubernetes’ vertical pod autoscaler. Additionally, we plan to support more cloud resource options such as memory, storage and network bandwidth, in that order. This should be a feasible task since the same assumptions and conditions regarding the variable demands would apply to them. Last but not least, we plan to unify the profiling, prediction model training and most accurate
model picking in a single, automated process. This way, an application’s developers would simply plug-and-play the runtime predictor to our elastic allocator without manually doing the work we did.
Bibliography


[13] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic re-
source scaling for cloud systems. In 2010 International Conference on Network

computing: What it is, and what it is not. In 10th International Conference
on Autonomic Computing (ICAC 13), pages 23–27, San Jose, CA, June 2013.
USENIX Association.

[15] Franck Houssen. Arpack-ng, a collection of fortran77 subroutines designed
to solve large scale eigenvalue problems, 2021. https://github.com/
opencollab/arpack-ng.

//github.com/kubernetes/community/blob/master/contributors/
design-proposals/autoscaling/vertical-pod-autoscaler.md.

manage-resources-containers/.

[18] Kubernetes.io. Kubernetes, an open-source system for automating deploy-
ment, scaling, and management of containerized applications, 2021. https://
kubernetes.io/.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel,
M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,
D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn:
index.html.

[20] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloud-
scale: Elastic resource scaling for multi-tenant cloud systems. In Proceedings
of the 2nd ACM Symposium on Cloud Computing, SOCC ’11, New York, NY,
USA, 2011. Association for Computing Machinery.

[21] Meng Tang, Mohamed Gadou, Steven Rennich, Timothy A. Davis, and Sanjay
Ranka. Optimized sparse cholesky factorization on hybrid multicore architect-

[22] Meng Tang, Mohamed Gadou, Steven C. Rennich, Timothy A. Davis, and
Sanjay Ranka. A multilevel subtree method for single and batched sparse
cholesky factorization. In Proceedings of the 47th International Conference
on Parallel Processing, ICPP 2018, New York, NY, USA, 2018. Association
for Computing Machinery.
