Policy-Aware Parallel Execution of Composite Services

Mai Xuan Trang
Department of Social Informatics
Kyoto University, Japan

Policy-Aware Parallel Execution of Composite Services

Abstract

The goal of this thesis is to enhance efficiency of using parallel execution in web service environments. To this end, we focus on parallel execution policies of web services to design a policy-aware parallel execution control architecture for composite services. Service-oriented architecture (SOA) has been a widely accepted and engaged paradigm for the realization of business processes that incorporate several distributed, loosely coupled partners. It allows users to combine existing atomic services to define composite services that meet users' requirements. With the expansion of SOA, the academic world and industry have started to adopt Web service and SOA to build applications. One example is natural language processing area; many language resources have been provided as language services. Some problems in language processing such as translation or morphological analysis of a large document often require long processing time. This type of service is related to the concept of data-intensive service. To reduce the processing time, efficient parallel and concurrent technology are promising approaches. Parallel execution is well studied in high performance computing (HPC) area, where people own computing resources and their tasks' implementation. Many existing parallel execution models introduce several

factors that cause performance limitation of parallel execution such as serial porting in the task, parallel overhead and balancing issues. These models help task providers analyze their programs and adjust the implementation or computing resources accordingly to improve parallel execution of the tasks. However, in SOA, service users do not own services. If there is a parallel execution limitation of a service, service users have no way to know the exact reasons causing the limitation. They may regard the limitation as a parallel execution policy of the service. Modeling parallel execution effects of web service as a policy model and considering policy of each atomic service to predict optimal parallelism for a composite service have not been considered in existing researches. Parallel execution policies of services affect performance improvement of composite service, which is a combination of the different services. To attain optimal performance improvement, users need to configure an optimal degree of parallelism (DOP) for the composite service regarding different policies of atomic services.

Therefore, in this thesis, we present following contributions toward the design of a policy-aware parallel execution control architecture for composite services:

1. Modeling parallel execution policies of web services:

   We propose a model to capture performance improvement behaviors of web services under parallel execution from the view of service users. The model need to correctly capture the parallel execution policies and need to be simple to be used in calculating performance of composite services. To define the model, first we analyze and observe performance improvement patterns of many web services when using parallel execution. From the observed performance improvement

patterns, we define a linear model to capture these patterns. Three types of policies are defined: Slow-down policy, restriction policy and penalty policy. We evaluate our proposed model by comparing with other regression model, the evaluation results show that our model accurately captures parallel execution policies of web services.

2. Predicting parallel execution performance of composite services:

   We propose a prediction model to predict parallel execution performance of composite services regarding different parallel execution policies of atomic services. The model embeds policy model to define different formula, calculating performance of composite services, for different workflow structures of the composite services. Four workflow structures are considered in the proposed model: Sequential structure, parallel structure, conditional structure and loop structure. Evaluation is conducted on real-world translation web services, the results show that our proposed prediction model has good prediction accuracy with regard to identifying optimal degree of parallelism of composite services.

3. Controlling parallel execution of composite services:

   Using the proposed prediction model we design an architecture for controlling parallel execution of composite services. This architecture can serve as a middleware for SOA platforms to support and control parallel execution of workflows. Typically, we implement this architecture as an extension of the Language Grid platform. When user invoking a composite service, the architecture able to analyze parallel execution policies of atomic services from binding information, it predicts the optimal DOP for the composite service. Then, it creates an optimal parallel execution deployment for the composite service. We extend the existing workflow execution engine to interprets parallel execution deployment and execute the composite service. In multiple users environment, the architecture can detect the multiple usage of an atomic service in different workflows. If these workflows are invoked in parallel, using prediction model, the architecture allocates a suitable parallel processes for each workflow in order to attain optimal reduction of execution time for all workflows.

Acknowledgements

First and foremost I would like to express my deepest gratitude to my supervisor, Professor Toru Ishida, who has been a tremendous mentor for me. None of the work in this thesis could have been carried out without his continuous guidance, valuable advice, fruitful discussions, and heartwarming encouragements. I would like to thank professor for encouraging my research and for allowing me to grow as a research scientist. His advice on both research as well as on my future career is priceless.

I also owe my sincere gratitude to my adviser committee members, Professor Masatoshi Yoshikawa, and Professor Yasuo Okabe for serving as advisers to keep monitoring my research progress and providing useful comments and suggestions.

I would especially like to thank Associate Professor Yohei Murakami for his valuable advices and close support to my research. I have been very appreciated by his so much effort in guiding my research.

I would like to show my gratitude to all faculty members of Ishida and Matsubara Laboratory, Associate Professor Shigeo Matsubara for his kind support, Assistant Professor Donghui Lin for fruitful discussion for my papers, Takao Nakaguchi for his advices in implementing technical parts of my PhD work, and Masayuki Otani for his useful advice for my presentations at lab seminars. I also greatly appreciate the laboratory coordinators, Ms. Terumi Kosugi, Ms. Hiroko Yamaguchi, and Ms. Yoko Kubota for their help in administrative affairs.

Special thanks also go to all my lab mates: Ari Hautasaari, Andrew W. Vargo, Huan Jiang, Chunqi Shi, Mairidan Wushouer, Amit Pariyar, Kemas Muslim Lhaksmana, Xin Zhou, Shinsuke Goto, Hiroaki Kingetsu, Xun Cao, Wenya Wu, Nguyen Cao Hong Ngoc, Arbi Haza Nasution, Mondheera Pituxcoosuvarn, Victoria Abou Khalil, Shohei Hida, Akihiko Itoh, Hiromichi Cho, Kaori Kita, Daisuke Kitagawa, Jun Matsuno and many others. I am happy for being a part of this wonderful lab with wonderful people.

This thesis is presented to my beloved wife Dang Thi Thanh Nga who spent a lot efforts for being a good partner of me in my daily life and was always my support in the moments when there was no one else can help, my two beloved daughters Mai Bao Ngan and Mai Bao Tran for their charming smiles and refreshing hugs. Words cannot express how grateful I am to my parents, Mai Xuan Hung and Trinh Thi Thuc for all of the scarifies that they have made on my behalf. Many thanks to other family member for their support and encouragement. I am also thankful to my friends in here Kyoto especially members of KYOTO RONG HOUSE who helped me to make my stay in Kyoto a very pleasant one.

My stay in Kyoto University was supported by JICA under SEED-Net scholarship from October 2012 to September 2015. This research was partially supported by Service Science, Solutions and Foundation Integrated Research Program from JST RISTEX, and a Grant-in-Aid for Scientific Research (S) (24220002) from Japan Society for the Promotion of Science.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Recent years have seen an emergence of service-oriented applications, where applications are created using existing services. Web service becomes a major trend in both academic and industry for loosely couple service-oriented architecture (SOA) and interoperable solutions across heterogeneous platforms and systems. SOA has been a widely accepted and engaged paradigm for the realization of business processes that incorporate several distributed, loosely coupled partners. It allows users to combine existing services to define composite services that meet users' requirements. With the expansion of SOA, the academic world and industry have started to adopt Web service and SOA to build applications. As in the natural language processing area, many language resources (e.g. machine translator, morphological analysis) have been provided as language services in service-oriented frameworks such as the Language Grid [Ishida, 2011] and PANACEA [Bel, 2010]. With the increasing volume of data to be analysed, one of the challenges in SOA is to make web service efficient in processing large-scale data. Data-parallel tasks is often used when a service process large amounts of data. One typical example of web services that use data-parallel task is translation task of a large data set. The input data set is split into independent partitions and multiple partitions will be translated by a translation service in parallel. This type of parallel execution would helps to decrease the overall execution time of the translation task. This thesis focuses on enhancing processing efficiency of data parallelism for language services such as translation services.

In high performance computing (HPC) area, there are some existing parallel execution performance models introducing factors that cause performance limitation of parallel execution. Most of models were defined from the view of tasks' providers who own the tasks and computing resources. These factors are mostly controlled and can be adjusted by the task providers to gain better parallel execution efficiency, such as serial portions in the task as pointed out in Amdahl's law [Amdahl, 1967, Sun and Chen, 2010], parallel overhead [Martin et al., 1997] and balancing issues. In SOA, due to some reasons, service providers may have some decisions to control parallel execution of their provided services. For example, if a provider is rich in computing resources, he/she may ready to accept large numbers of concurrent requests, if the provider have a limit computing resources or due to some security concerns he/she may control the number of concurrent requests that their provided services can served. However, service users do not own the services, nor they can control the services' execution or computing resource, they have no way to know the exact reasons that cause the parallel execution limitation of the web services. Service users only observe performance improvement patterns of the services. In this thesis, from the point of view of service users, we regard the performance improvement patterns as policies of service providers. Some providers may explicitly describe policies for their provided services, for example Google describe policies about per-user limit on their website[1]. Different from HPC or parallel computing areas, in SOA, service users do not own services, provided by other providers, they cannot change the services' implementation or providers' computing resource to have better parallel execution. In order to use parallel execution efficiently, users need to adapt their invocation according to the services' behaviors (we defined as services' policies in this thesis).

Composing and optimizing composite services, or workflows of atomic services, has gained increasing attention in SOA. Technologies such as data-intensive and many-task computing [Raicu et al., 2012, Humbetov, 2012], and scientific workflows [Taylor et al., 2014] have the potential to enable rapid data analysis for workflow. Many studies have proposed parallel and pipelined execution techniques to speed-up workflows [Pautasso and Alonso, 2006], and adaptive parallel execution approach for workflows in cloud environments regarding the availability of resources [Oliveira et al., 2012]. There are also many methodologies that focus on creating composite services which can attain the optimal Quality of Service (QoS) based on linear integer programming [Cardoso et al., 2004, Zeng et al., 2004] or Genetic Algorithms (GAs) [Canfora et al., 2008]. Most of the existing researches do not deal with the parallel execution tech-

[1]Google per-user limit policy:https://developers.google.com/analytics/devguides/reporting/core/v3/limits-quotas

nology in web service composition. Neither do they consider how the parallel execution policy of each atomic service affects the efficiency of the execution of the composite service.

1.2 Objectives

The objectives of this thesis is to design a policy-aware parallel execution control architecture for composite services. The target is to enhancing processing efficiency of data-parallelism for language services such as translation services. Since web services are provided by different providers with different policies, the system needs to analyze parallel execution policies of atomic services that form a composite service. Based on these policies and control structure of the composite service, the system predicts performance and determines the optimal parallelism for the composite service. The system will maintain the parallel execution of composite services in order to attain optimal performance improvement. There are two motivations to achieve these goals:

1. Help service users to have better data-parallelism (parallel execution) when invoking web services. Parallel execution policies of web services are normally not described explicitly by service providers, service users can only observe performance improvement patterns of web services when using parallel execution. However, there is no existing work on modelling these patterns. In this thesis, we first focus on modeling the performance improvement patterns as parallel execution policies of web services. The model (hereafter called the "policy

model") should correctly capture parallel execution effects of atomic services, and should be simple to be used in predicting performance of composite services. The policy model is useful for service users to understand parallel execution policies of web services. This enables users to adapt parallel invocations of a web service to the service policy in order to attain the optimal speed-up.

2. Help SOA platforms to support better parallel execution for composite services (workflows) considering policies of all atomic services. We propose a prediction model that embeds parallel execution policies of all atomic services that compose a composite service to calculate parallel execution performance of the composite service. From the calculation, the optimal degree of parallelism (DOP) of the composite service, where it attains optimal performance improvement, is determined. Using the proposed model, a parallel execution control architecture is design as a middle-ware to help SOA platforms to maintain optimal parallel execution of composite service.

1.3 Issues and Approaches

In designing and implementing the policy-aware parallel execution control architecture for composite services we listed three approaches to deal with following issues.

1. **Modeling parallel execution policies of atomic services.** There are many factors that can have effects to parallel execution performance improvement of web services. The limitation of parallel execution

may because of limitation of computing resources, limitation of the service's implementation, or decision of service providers. However, service users do not have clear information about the reasons of the limitation. Instead, they have a clear image about the performance improvement pattern of the service. Users also only care about this pattern in order to have suitable parallel execution of the service. From observation of performance improvement patterns of different services, we define a policy model to capture these performance improvement patterns of different atomic services. The word "*policy*" here means how an atomic service reacts to parallel execution, it does not necessary mean a policy employed by the service provider. The policy model is designed to be simple in order to be used in predicting performance of composite services latter on. It also needs to accurately capture the policy patterns.

2. **Predicting performance of composite services.** As a composite service is created by combining different atomic services provided by various providers. Parallel execution policies of different atomic services may vary. A question arises : "which is the optimal degree of parallelism for a composite service, where it attains optimal performance improvement?". We propose a model that uses the policy model and characteristics of the composite service control structures to form formulate for predicting the service's performance. From the prediction, we can have an idea which is the optimal value of the degree of parallelism (or DOP).

3. **Controlling parallel execution of composite services.** After predicting performance of composite services, the system needs to con-

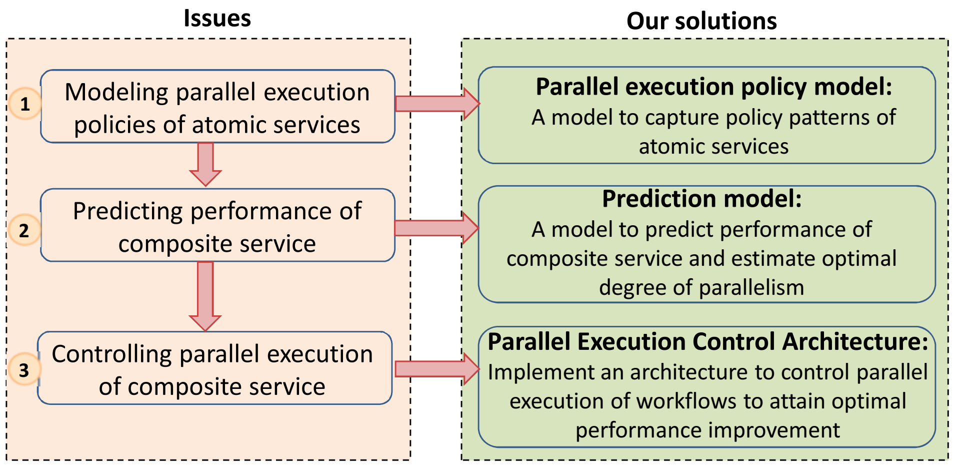

Issues
Our solutions
1
Modeling parallel execution policies of atomic services
2
Predicting performance of composite service
3
Controlling parallel execution of composite service
Parallel execution policy model:
A model to capture policy patterns of atomic services
Prediction model:
A model to predict performance of composite service and estimate optimal degree of parallelism
Parallel Execution Control Architecture:
Implement an architecture to control parallel execution of workflows to attain optimal performance improvement

Figure 1.1: Overview of solutions for the policy-aware parallel execution control architecture

trol parallel execution of composite services for the optimal outcome (minimize execution time). Since atomic services can be dynamically bound to a composite service, the system needs to dynamically adjust DOP of the composite service based on binding information. Furthermore, in multiple users environment, an atomic service can be used in multiple composite services, if these composite services are invoked in parallel, in order to maintain optimal outcome, the system allocates suitable number of concurrent processes for each composite service based on the atomic service's policy.

As shown in Figure 1.1, to realize the policy-aware parallel execution control architecture, this research starts from proposing the policy model to capture policy patterns of atomic services. Using the policy model we introduce a model to predict performance of composite service and estimate the optimal degree of parallelism of composite service. At the end, we focus on dealing with implementation issues to implement the policy-aware architecture that can control parallel execution of composite services to attain the optimal execution time reduction.

1.4 Thesis Outline

This thesis consists of six chapters including Chapter 1. The content of each of the remaining chapters are summarized as follows.

Chapter 2 describes the background of this thesis. This chapter begins with a general introduction of service composition and quality of service (QoS), and presents previous work on parallel execution of web services to improve the services' processing efficiency. In order to create an overview on parallel execution in web service environments, this chapter classifies the existing work into three groups according to the parallelization techniques found in Workflow Management Systems (WfMSs): data parallelism, task parallelism, and pipeline parallelism.

Chapter 3 depicts a way to model parallel execution policies of atomic services. In an environment where service providers employ policies that arbitrarily limit parallel execution of their services, service users' excess parallel execution of the services decreases the processing efficiency of their whole set of tasks. Therefore, the service users need to know the optimal degree of parallelism in order to maximize the processing efficiency of the tasks before invoking the services. To this end, by measuring the effects of the degree of parallelism on processing efficiency of more than 50 services (hereafter atomic services), this chapter classifies the parallel execution policies into three types: low acceleration policy that gradually decreases the degree of speedup, steady policy that fixes the processing efficiency when the degree of parallelism is beyond a certain amount, and penalty policy that decreases the processing efficiency as the degree of parallelism increases. Moreover, this chapter also details a way to capture parallel execution processing efficiency of atomic services that employ a combination of different policies. A series of experiments on 50 different atomic services shows that the proposed model can estimate the processing efficiency of parallel execution with a lower standard error than the existing curve fitting with linear and quartic regression.

Chapter 4 presents a way to predict the processing efficiency of composite services with parallel execution. A composite service is a service where a workflow orchestrates atomic services with different policies. In order to maximize the processing efficiency of composite services with parallel execution, service users need to optimize the degree of parallelism by considering policies of all atomic services. Therefore, this chapter introduces data parallelism and pipeline parallelism to execute a workflow in parallel. Processing time-line of the pipeline parallelism is used to define an aggregation function to compute processing efficiency for each simple workflow consisting of single control construct, such as a sequential construct, concurrent construct, conditional construct, and loop construct. This chapter also proposes a method to synthesize the policies of atomic services in a complex workflow consisting of an arbitrary combination of control constructs. Finally, this chapter evaluates accuracy of the proposed method in predicting the optimal degree of parallelism of composite services. A series of experiments on composite services combining several different translation services shows that the proposed method has good prediction accuracy in identifying optimal degrees of parallelism for composite services. The proposed method is helpful in designing architecture to control parallel execution of composite services for improving processing efficiency.

Chapter 5 proposes a service platform architecture to control parallel execution of composite services based on parallel execution policies of atomic services. Using the proposed method of predicting optimal degree of parallelism of composite services, this chapter designs architecture for controlling parallel execution of composite services. The architecture first analyzes parallel execution policies of atomic services that compose the composite service. It then computes the optimal degree of parallelism of the composite service. Finally, the architecture generates a parallel execution configuration file that is interpreted by an extended workflow engine to control parallel execution of the composite service. Furthermore, in order to efficiently process multiple workflows that share the same atomic service, the architecture re-calculates optimal degree of parallelism of each workflow by considering multiple requests from different workflows sent to the shared atomic service. It then updates the parallel execution configuration files with the new optimal degree of parallelism for the workflows at run-time. To verify the effect of the architecture in maintaining optimal parallel execution efficiency of workflows, the architecture is implemented in the Language Grid, a service platform that is specialized in natural language processing. An experiment is conducted on multiple language composite services; the results show that the proposed architecture can significantly improve the processing efficiency of parallel execution of composite services.

Finally, we conclude the thesis in Chapter 6 by summarizing the results obtained through this research and addressing future works.

Chapter 2

Background

2.1 Service-Based Application

Over the recent years, the concept of Service-Oriented Computing (SOC) has emerged in order to address the challenges posed by the increasing complexity of heterogeneous software applications and the need of these applications to be integrated within and across organisational boundaries. SOC provides the theoretical foundations to address these challenges by utilising services as fundamental building blocks for developing applications [Papazoglou, 2003]. A service is a reusable component that is loose-coupled, well-defined, self-described and agnostic of any particular deployment platform and/or implementation technology.

The idea that providers can offer their applications' functionalities or services over the Web, and other consumers can use and compose these services, has led to the emergence of a new architectural style, that is common-

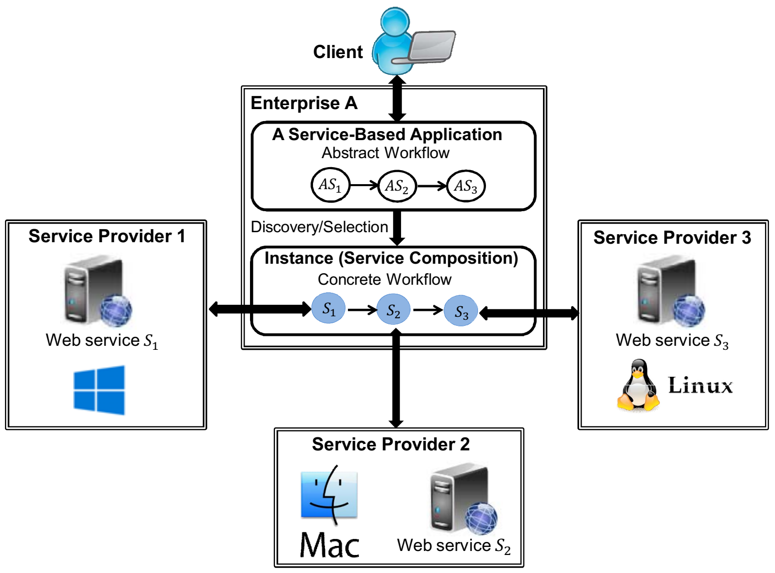


Figure 2.1: Service-Based Application

ly referred to as, *Service-Oriented Architecture* (SOA). With this architecture, the development of a distributed application is performed by creating a Service-Based Application (SBA). A Service-Based Application is a composition of independent services which perform the desired functionalities. The running application is performed by dynamic binding to concrete services. Concrete services could be provided by third parties, not necessarily by the owner of the service based application.

Figure 2.1 illustrates an example of SBA. Enterprise A intends to develop a service-based application in order to provide a certain functionality for its clients. Firstly, an abstract workflow is defined by decomposing the expected functionality into a collection of interrelated tasks (or activities) that function in a logical sequence to achieve the ultimate business goal. Such an abstract workflow is not executable because it lacks binding references. Therefore, the abstract workflow is required to be instantiated before the execution can start. The instantiation of workflow aims to construct an executable *concrete workflow* by mapping each task AS_i to a specific service.

A concrete workflow presents a service composition which binds the execution of each abstract service AS_i to a specific service S_i. In this context, all the services involved in a service composition are defined as *constituent services*. As shown in Figure 2.1, al the constituent service can be developed based on different technology, provided by different providers and running on heterogeneous platforms.

The advent of cloud computing promotes such Software-as-a-Service (SaaS) model. Service providers do not have uniform conditions, they may provider services with different policies. Some provider may offer service packages which allow unlimited invocations within a certain period, others may not allow users to invoke their service with many concurrent processes. Such different policies are needed to be considered when constructing a service-based application (composite service). Additionally, the SBA exhibits the following characteristics:

- **Distributed resources.** The composition of services represents distributed computational resources, from both software level and hardware level. SBA users will have no control over computational resources and implementation of each service.

- **Cross administration domain.** Services can be developed and managed by different organizations/enterprises (service providers). SBA users need to follows policies of service providers.

Next, we introduce some basic concepts for service-oriented computing, which are used in this thesis.

2.1.1 Web Service

Web service is the basic building block for developing SBA. It is a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application. Web services constitute a distributed computer infrastructure, made up of many different interacting application modules trying to communicate over private or public networks to virtually form a single logical system.

A more precise definition is published by W3C[1]:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

From the above description, a web service provides an interface defined in terms of XML messages and that can be accessed over the Internet. It is an application that exposes a function which is accessible using standard Web technologies and that adheres to Web services standards, to name a few, Extensible Markup Language (XML) [Bray et al., 1998] for presenting data, Simple Object Access Protocol (SOAP) [Gudgin et al., 2003] for transmitting data, the Web Service Description Language (WSDL) [Chinnici et al., 2007] for defining services in-

[1]W3C-the World Wide Web Consortium: http://www.w3.org/TR/ws-arch/

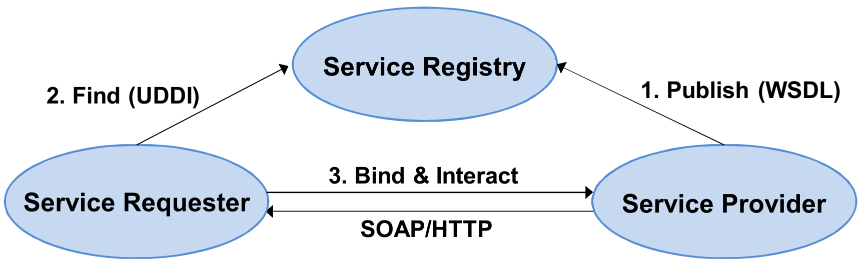


Figure 2.2: Web Service Architecture

terfaces etc.

Web Service Architecture. Figure 2.2 shows the Web Service Architecture which consists of three main parts: service provider, service requester and service registry.

- A service provider implements web services and defines the web services' interfaces to publish to a service registry.
- A service requester invokes the required service from the service registry
- Service registry stores the published services in a repository form.

Quality of Service. The Quality of Service (QoS) reflects the non-functional performance of a Web service. A QoS attribute indicates how well a Web service performs in terms of a specific quality metrics. We list some examples of QoS attributes as following. More detail of the QoS attributes are described in [Al-Moayed and Hollunder, 2010].

- **Time.** The response time (execution time) reflects the expected duration for a service finish processing a task.
- **Cost.** The cost is the fee that a service requester has to pay in order to

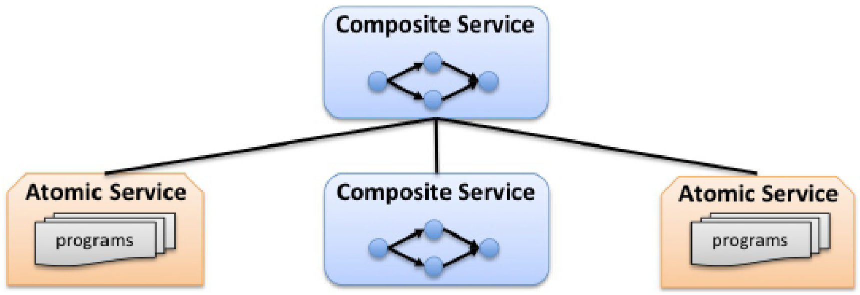


Figure 2.3: Hierarchy of Composite Services

invoke a Web service.

- **Availability.** The availability reflects the probability that the Web service is available.

Atomic Web Service and Composite Web Service. According to different implementation, a service can be classified into two categories: an atomic service or a composite service. The former represents an individual software component which provides the expected functionality; whereas the latter defines a service composition which aggregates a number of basic and fine-grained services.

2.1.2 Web Service Composition

One of the key driving ideas in SOC is service composition, which is the possibility to quickly crate new services and applications (composite services) by combining the existing ones (component services). The need for service composition arises when current existing services do not provide the required functionality or when the coordinated execution of existing services is needed in the context of some new business objective. Usual-

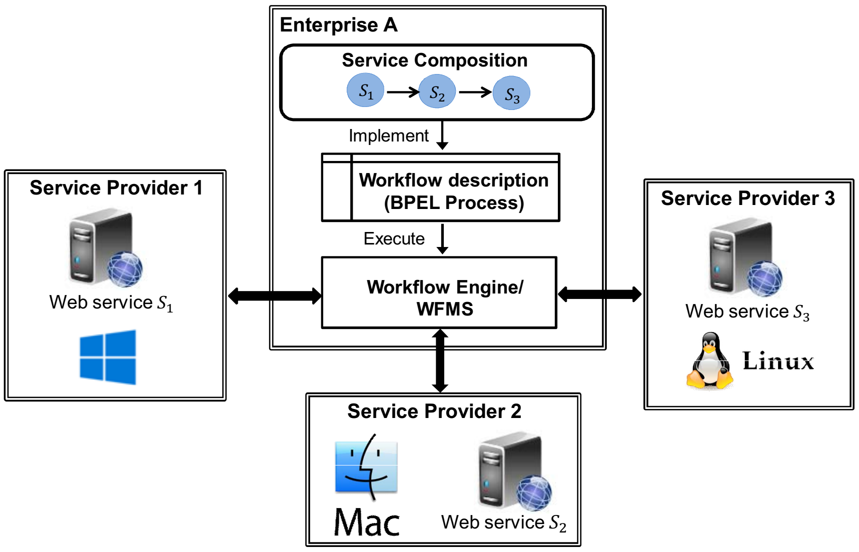


Figure 2.4: Workflow Orchestration

ly, the process of service composition is recursive, i.e., the newly created composite service can consequently be used as component services in other compositions as shown in Figure 2.3. Web service composition can be modelled as executable process using Web service composition languages like BPEL [Andrews et al., 2003] and WSFL [Leymann et al., 2001] incorporating the concepts and mechanisms in the workflow community. Therefore, the logic in a composite service can be captured using workflow patterns [van Der Aalst et al., 2003]. The composite web service provider typically runs an orchestrator (Workflow engine/Workflow Management System(WFMS)) that invokes the aggregated services according to a predefined workflow as shown in Figure 2.4. Since the workflow is defined based on unambiguous functionality description of a service ("*abstract service*"), and several alternatives ("*concrete services*") may exist that match such a description [Preist, 2004], the workflow engine/WFMS need to able to find a suitable concrete service.

QoS-Aware Service Composition. In order to select suitable concrete services for an abstract composite service many approaches have been proposed based on QoS. The stage of finding a optimal set of concrete web services that satisfies several QoS constraints (e.g., minimising the overall response time) is called QoS-aware service optimization. This problem has been addressed by many studies from different angles. In [Zeng et al., 2003], Zeng et al. focused on the runtime management of service composition under dynamically changeing QoS environments, and the optimizations were carried out to avoid global constraint violations at runtime. Many works used constraint satisfaction problem (CSP) to form an optimal composite service. Guan et al. [Guan et al., 2006] proposed one of the first framework for QoS-Aware service composition. Their framework modelled the functional requirements as hard constraints and used constraint hierarchies as a formalism for specifying QoS constraints. Hassine et al. [Hassine et al., 2006] proposed approached using constraint optimization problem formalism to determine the best executable workflow according to predefined optimality criteria. Jaeger et al. [Jaeger et al., 2004] proposed an approach for calculating the QoS of a composite service based on the well-know workflow patterns proposed in [van Der Aalst et al., 2003]. Based on this calculation, Yu et al. [Yu et al., 2007] proposed a set of algorithms for Web service selection with end-to-end QoS constraints. An important stage in QoS-Aware service composition is QoS computation, many studies have introduced different factors in QoS computation. Bramantoro and Ishida [Bramantoro and Ishida, 2009] introduced a method to include user preferences and skill into QoS metrics. Goto et al. [Goto et al., 2011] proposed an idea to use reputation as an QoS factor to select services for a composite. To deal will context-aware QoS, where the QoS of service may vary in different context, Lin et al. [Lin et al., 2012] proposed a dynamic service selection approach based on context-aware QoS. However, to the best of our knowledge there is no existing work that consider providers' decision on parallel execution of atomic services in service composition.

2.2 Parallel Execution in Service Workflow Environments

2.2.1 Workflow Management Systems

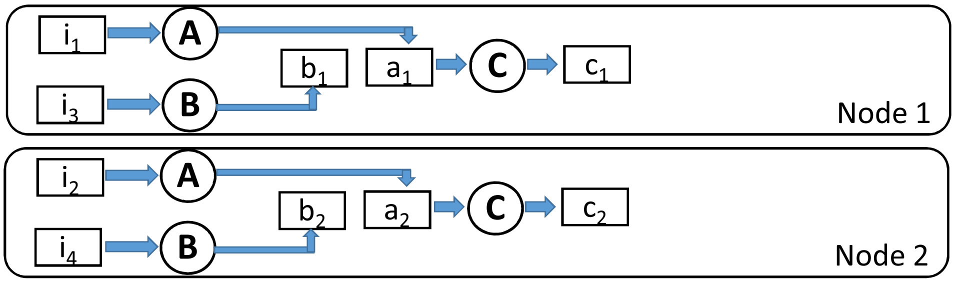
Composite services are normally composed, managed and executed by a workflow management system (WFMS). During the last years many WfMSs appeared; to name only a few of the most popular ones: Triana [Allen et al., 2003], Kepler [Altintas et al., 2005], Pegasus [Deelman et al., 2005], KNIME [Berthold et al., 2008], Galaxy [Goecks et al., 2010], and Taverna [Missier et al., 2010]. Each WFMS has its own workflow language to describe workflow composition. The languages come along with a workflow engine which interprets the workflow description and invoke the component services. To help users to design workflows, a WFMS may also provide a graphical user interface. As scientists and engineers are using WFMSs to build more and more complex workflows to manage and process large data sets especially for scientific data. This trend requires current WFMSs to support parallelization to reduce execution time of workflows.

2.2.2 Parallel Execution of Workflows

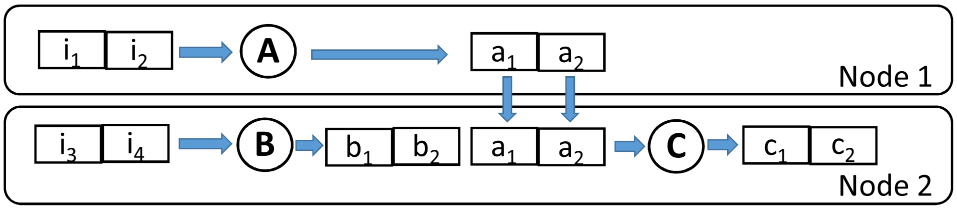
Workflow parallelization identifies the tasks that can be executed in parallel in the workflow execution plan (WEP). There are different means to accomplish parallelization, all of which involve subdividing either the set of workflow tasks or the input data (or both). According to the dependencies defined in a workflow, different parallelization techniques can result in various execution plans. Some parameters can be used to evaluate the efficiency of each technique. An important parameter of parallelization is the degree of parallelism, which is defined as the number of concurrently running computing nodes or threads at any given time and that can vary for a given workflow depending on the type of parallelism. In this section, we distinguish three major types of parallelization: data parallelism, task parallelism and pipeline parallelism. Data parallelism deals with the parallelism within a task while task parallelism and pipeline parallelism handle the parallelism between different tasks.

Data Parallelism

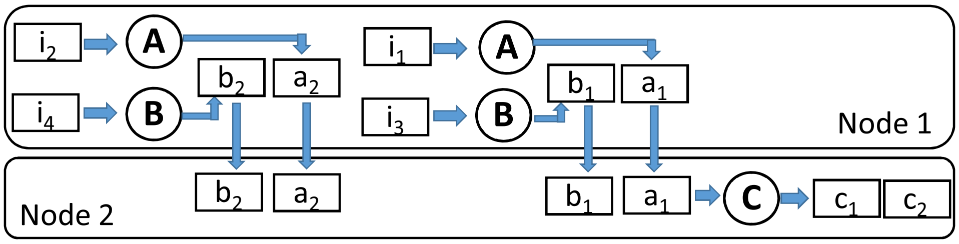
In data parallelism, input or intermediate data is split into distinct partitions, each of which is processed on a different compute node (or thread). This means that the workflow (or part of the workflow) is replicated on each compute node for a different partition of data. Data parallelism is only feasible if data can be split into independent partitions. Most suitable for data parallelism are so-called embarrassingly parallel problems in which data items can be processed independently from each other. Figure 2.5a shows and example when the input data is split into two independent partitions, the two partitions can be processed in parallel with two instances of the workflow



(a) **Data parallelism.** The workflow is performed in two computing nodes simultaneously. Each computing node processes a data partition.



(b) **Task parallelism.** A and B is performed in two computing nodes concurrently. C begins execution after execution of A and B



(c) **Pipeline parallelism.** C starts execution once a data partition is ready. When A and B are processing the second part of data (i_2, i_4), C can process the output of the first part (a_1, b_1) at the same time

Figure 2.5: **Different types of parallelism.** Circles represent tasks. There are three tasks: A, B and C. A and B are independent. C processes the output data produced by A and B. Rectangles represent data partitions.

in two compute nodes. As the input data needs to be partitioned, e.g., by a partitioning task, the activity result is also partitioned. Thus, the partitioned output data can be the base for data parallelism for the next tasks. However, to combine the different results to produce a single result, e.g. the final result to be delivered to the user, requires special processing, e.g., by having all the tasks writing to a shared disk or sending their results to a task that produces the single result. This parallelism technique is also known as the strong scaling technique.

Task Parallelism

Task parallelism is achieved when the tasks composing a workflow are executed in parallel over several computing nodes. It is only applicable to tasks located on parallel branches of the workflow. The maximum number of parallel tasks can be computed easily in advance by analyzing the workflow structure. However, choosing the right scheduling strategy is not trivial, as parallel tasks might exhibit varying execution times. Differences in task execution times are also the reason for junctions in the data flow being difficult to handle: tasks requiring input data from several concurrent previous tasks have to wait until all parent tasks have finished execution. Buffering of intermediate results or synchronization of task execution is therefore required for task parallelism. Figure 2.5b shows an example of two tasks in a workflow can be executed in parallel when they are independent.

Pipeline Parallelism

In pipeline parallelism, sequential steps of data processing are executed simultaneously on different parts of input data. Thus, one part of the output data of one task is consumed intermediately by the next dependent task in a pipeline manner. Figure 2.5c shows an example of pipeline parallelism. Pipeline parallelism shares characteristics with task and data parallelism and can be considered a subset of both:

- Similar to data parallism, input data is split and processed independently of different compute nodes. However, in contrast to data parallelism, the chronological order in which fragments of data are processed as well as the assignment of tasks to compute nodes are determined in advance and restricted by the concept of the pipeline.
- As with task parallelism, the tasks composing a workflow are distributed over independent compute nodes. However, it is not limited to tasks on parallel branches of the workflow, resulting in a potentially higher degree of parallelism.
- Pipeline parallelism is closely related to streaming techniques [Gordon et al., 2006].

The efficiency of data parallelism and pipeline parallelism of a composite service can be improved by choosing a suitable degree of parallelism. In this thesis we focus on these two parallelism techniques and use parallel execution policies of atomic service to determine the optimal DOP of the composite service. We mostly use language services especially translation services for our analysis and experiments throughout the thesis.

Chapter 3

Modeling Parallel Execution Policies of Web Services

3.1 Introduction

Web service based applications are performing more and more, larger and larger transactions. They are becoming intensive users of web transaction through Service Oriented Architecture Standard. The accessing, transferring and processing of data need to occur parallel in order to tackle the problem brought on by the increasing volume of data. With the advent of high performance computing and cloud computing technologies, service providers enable to support parallel execution for their provided services.

Consider a client processing a large data by invoking a web service. Strong scaling technique (data parallelism) can be used to process the data faster by splitting the data into independent partitions and multiple partitions are sent to the service concurrently. If the service supports parallel execution, a typical customer would expect the speed-up of the client will be directly proportional to the number of concurrent requests (or the degree of parallelism - DOP). However, the actual achieved speedup is not always directly proportional to the DOP. There are several factors that effect efficiency of parallel execution such as serial fractions in the service implementation as pointed out in Amdahl's law [Amdahl, 1967, Sun and Chen, 2010], or parallel overhead [Tallent and Mellor-Crummey, 2009]. Service providers may also employ policies to limit parallel execution of their provided services based on their arbitrary decisions. Service providers have control over those factors, they can change some of the factors to control parallel execution support of provided services. However, from the view of service users, s-ince they do not own the service, they cannot know exact reasons causing the parallel execution limitation of the service. What the service users observe is the performance improvement behavior of the service when they invoke the service with parallel requests. In this context, we model the performance improvement behaviors as parallel execution policies of services. In this chapter, first of all we analyse and observe parallel execution effect-s of different atomic services. Then, from the observation we determine different performance improvement patterns and define the policy model. Finally, we evaluate how well our proposed policy model can capture parallel execution effects of different web services.

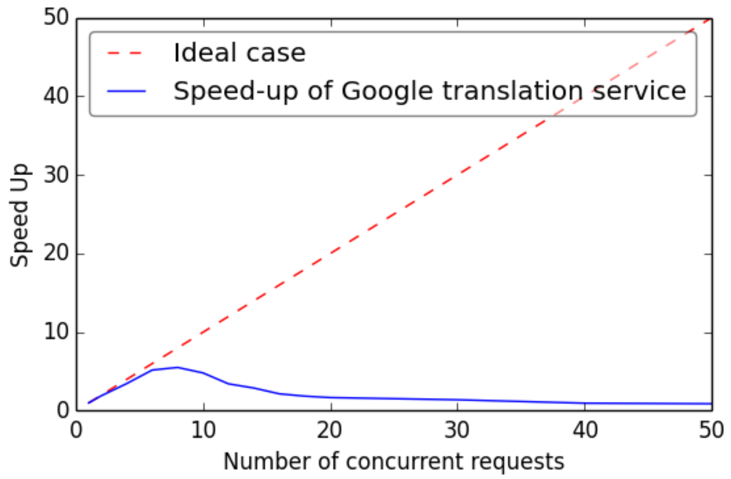


Figure 3.1: Ideal and actual speed-up cases

3.2 Motivation Example

Consider a translation application that uses Google translation service to translate a document. In order to reduce the translation time, an user configures the application to split the document into M independent partitions, and then send n multiple requests to Google translation service in parallel. Suppose that the method of splitting document is determined (M is fixed). Increasing n is expected to reduce the time taken to translate the whole document. Let *Speed-up* ($S(P)$) of the application be the ratio of the execution time of the application when $n = 1$ to the execution time of the application when $n = P$ ($S(P) = T(1)/T(P)$). A straightforward extrapolation to the higher number of concurrent requests would give the speed-up shown by the dashed line in Figure 3.1.

This type of extrapolation is too common and unwarranted in our experience. As we will see, the actual speed-up of the application is more likely to follow the solid line in Figure 3.1. The difference between these two predicted curves is significant. This example underscores the importance of obtaining a thorough understanding of the speed-up characteristics of a web service before invoking the services with parallel execution. One way to accomplish this is to assess speed-up patterns by analysing the parallel execution effects of different types of web services. Once these patterns are determined we can define a model which can help users to better estimate service performance under parallel execution.

3.3 Parallel Execution of a Web Service

We use *Data Parallelism* to perform parallel invocation to a web service as follows. Assume that a client wants to process a large dataset. At the client-side, the input data is split in to M partitions and n threads of the client are created to send n partitions to the service in parallel as shown in Figure 3.2. At server-side the service needs to serve n requests in parallel. Execution time required for processing the input data depends on the number of concurrent requests, denoted by $f(n)$.

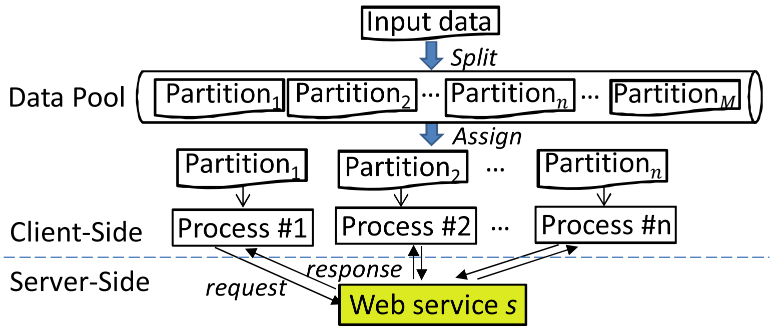


Figure 3.2: Parallel execution of a web service

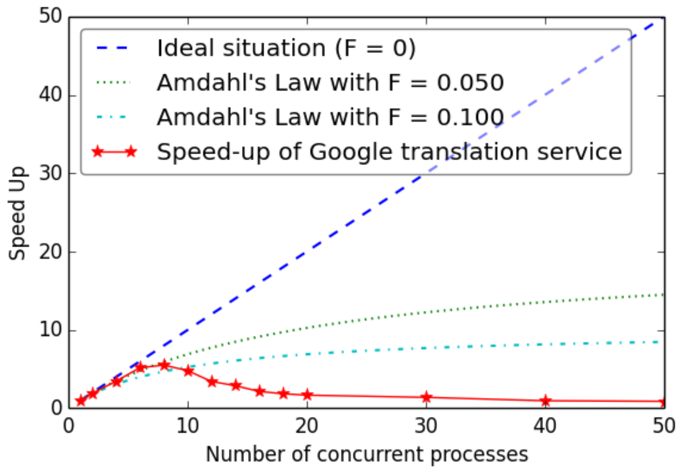


Figure 3.3: Different speed-up behaviors

Performance Speed-up

We use *Speed-up* as a measure of the reduction in execution time taken to execute a fixed workload when increasing number of concurrent threads. Speed-up is calculated by the following equation: $S(n) = f(1)/f(n)$, where $f(1)$ is the execution time required to perform the work with a single thread and $f(n)$ is the time required to performance the same task with n concurrent threads.

Possible speed-up behaviors of a service may fall into three categories:

- *Linear*–the speed-up ratio is equal to the number of concurrent processes, n, i.e., $S(n) = n$.
- *Sub-linear*–the speed-up ratio with n concurrent processes is lass than n, i.e., $S(n) < n$
- *Super-linear*–the speed-up ratio with n concurrent processes is greater than n, i.e., $S(n) > n$

Several models have been proposed to describe those speed-up behavior categories for parallel algorithms and architectures [Sun and Chen, 2010]. A well known and most cited model is Amdahl's law [Amdahl, 1967], which models the effect of the serial fraction of the task to the speed-up of the task. Different ratios of serial parts (F) yield different speed-up behaviors as shown by the dash lines in Figure 3.3.

Most of existing models assume that the performance speed-up is determined chiefly by task limitations, computing resource limitations or parallel overhead. With the improvement of high performance computing, these limitations may not be the problem for rich providers, such as Google or Microsoft. Instead, service provider's arbitrary decision about how to implement parallel execution (parallel execution policies) may have big effect on the performance speed-up. However, this kind of effect is not considered in existing models. Furthermore, considering from the view of service users, since they do not own services, they can only observe performance improvement behaviors of the services when using parallel execution. In this work we regard the performance improvement behaviors of services (such as the solid line in Figure 3.3) as parallel execution policies of the services. We will analyze and define a model for parallel execution policies.

3.4 Testing Methodology

In order to define a model for parallel execution policies, we first test many web services with parallel execution. We then observe and analyze the performance improvement pattern of each service.

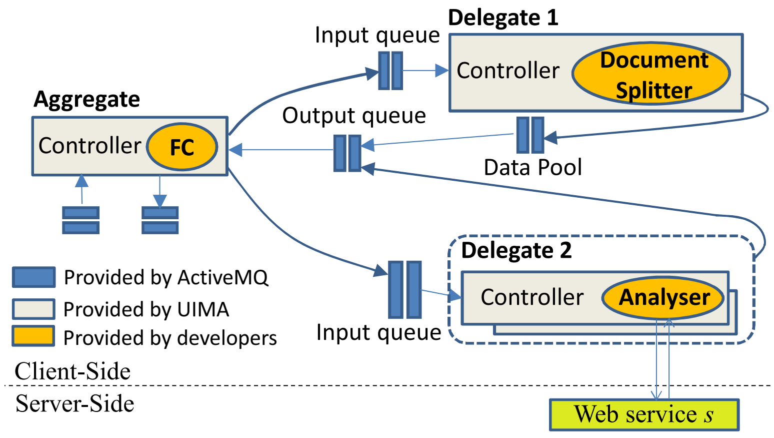


Figure 3.4: Implementation concept of the test system

The testing environment is based on a client-server architecture. We create a client to invoke web services with parallel execution. One challenge is to collect different web services provided by different providers for analysis. One of the most reliable sources we used is the Language Grid [Ishida, 2011]. The Language Grid (LG) provides an infrastructure for sharing and combining language services. Different groups or providers can join and share language services on the Language Grid[1]. Currently, more than 140 organizations have joined the Language Grid to share over 170 language services. We also assessed web services from outside the LG, such as from Programmableweb[2].

3.4.1 Experiment Implementation

We implement a client using multi-threading technique to invoke web services. First, the input data is split into independent partitions. Then, *n*

[1]Web services on the LG: http://langrid.org/service_manager/language-services
[2]ProgrammableWeb: http://www.programmableweb.com/

threads of the client are initialized to process *n* partitions in parallel. Therefore *n* requests are sent to the service concurrently. We also use pooling technique to stream data partitions to the client whenever a thread is available. We use Apache UIMA[3] to realize our test system. First, we create a Document Splitter to split input document into independent partitions and store partitions to a queue. UIMA uses ActiveMQ[4] to create and manage queues. We create a client which invoke a web service to process data partitions from the queue. We implement a Follow Controller (FC) to connect the Document Splitter and the Client, and control the queues and number of threads of the Client. Figure 3.4 shows implementation concept of our experiment in the UIMA framework.

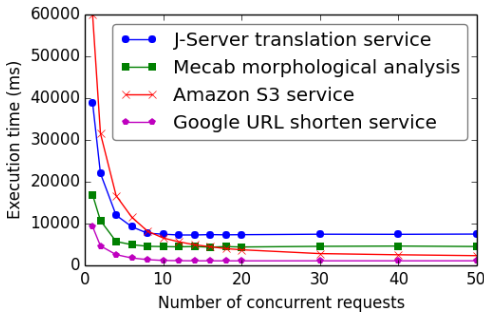
3.4.2 Performance Improvement Patterns

We conducted series of experiments on more than 50 web services provided by different providers, about two-thirds of them are registered in the Language Grid, the other are collected from outside the Language Grid. We observed different performance improvement patterns of different web services. These patterns are categorized as follows:

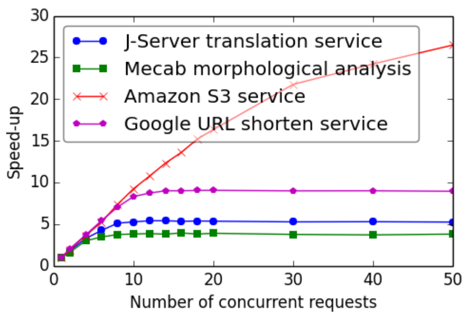
Slow-down and restriction patterns

Figure 3.5 shows slow-down and restriction patterns of several services. For example, performance improvement of J-Server translation service follows slow-down pattern when number of concurrent requests smaller than 16, the performance is throttled when the number of concurrent requests exceeds 4

[3]Apache UIMA: http://uima.apache.org/
[4]Apache ActiveMQ: http://activemq.apache.org/

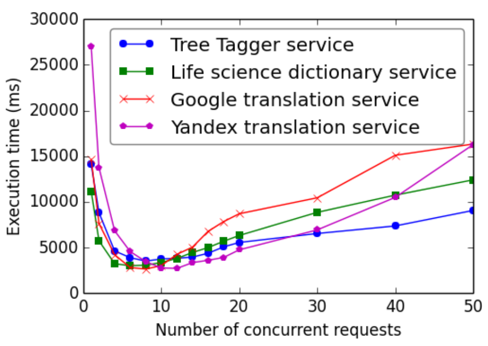


(a) Performance improvement

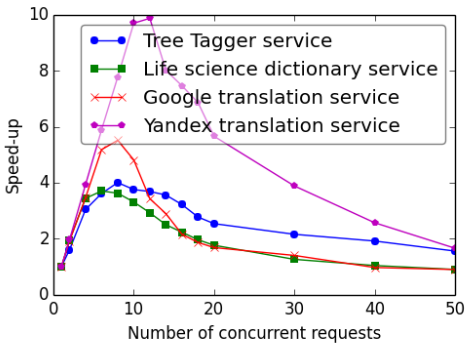


(b) Speed-up

Figure 3.5: Web services with slow-down and restriction patterns



(a) Performance improvement



(b) Speed-up

Figure 3.6: Web services with slow-down and penalty patterns

(slow-down point P_s). The performance statures when the number of concurrent requests reaches to 16 (restriction point P_r). Similarly, performance improvement of Mecab morphological analysis service follow slow-down and restriction patterns with $P_s = 4$ and $P_r = 14$. $P_s = 2$ and $P_r = 12$ are slow-down point and restriction point of Google URL shorten service. A-mazon S3 service shows a slow-down pattern with $P_s = 14$.

Slow-down and penalty patterns

Figure 3.6 shows slow-down and penalty patterns of several services. For example, performance improvement of Google translation service follows slow-down pattern when number of concurrent requests smaller than 8, the performance is throttled when the number of concurrent requests exceeds 4 (slow-down point P_s). The performance is reduced when number of concurrent requests exceeds 8 (penalty point P_p). Similarly, performance improvement of Yandex translation service follows slow-down and penalty patterns with $P_s = 10$ and $P_p = 12$. TreeTagger service shows slow-down and penalty patterns with $P_s = 4$ and $P_p = 8$.

3.5 Parallel Execution Policy Model

From the observation of performance improvement patterns of services under parallel execution, we define a model to capture those different patterns. We call this model is *parallel execution policy model* (or *policy model* in short). One main objective of this model is to help users to easily embed service policies into the calculation of composite service performance. The model should be simple to be used when calculating composite service performance. We define the policy model as a linear model, the definition of the model is as follows:

Definition (Parallel Execution Policy). *A parallel execution policy of a web service is defined by the change of the performance improvement trend of the service under parallel execution. This performance improvement trend is determined by a tuple of parameters (α, $\alpha^{\star}$, α', P), with each parameter is defined as follows:*

- *Suppose that the service processes M data partitions using parallel* *execution. The execution time of the service depends on number of concurrent processes (n) of the service, denoted by $f(n)$.*

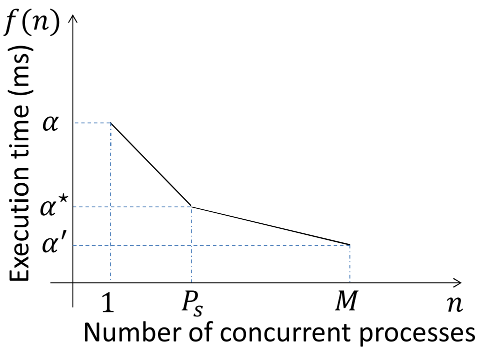
- α *is execution time of the service when the M partitions are serially executed, i.e.,* $n = 1$*:* $f(1) = \alpha$.
- *P is the upper bound of concurrent processes specified by the service provider, beyond which the performance improvement trend changes.*
- $\alpha^{\star}$ *is time taken by the service to process M partitions with P concurrent processes:* $f(P) = \alpha^{\star}$.
- α' *is time taken by the service to process M partitions with N (*$P < N \leq M$*) concurrent processes:* $f(N) = \alpha'$.

From the observed performance improvement patterns, we define three types of parallel execution policy as follows: Slow-down policy, restriction policy and penalty policy.

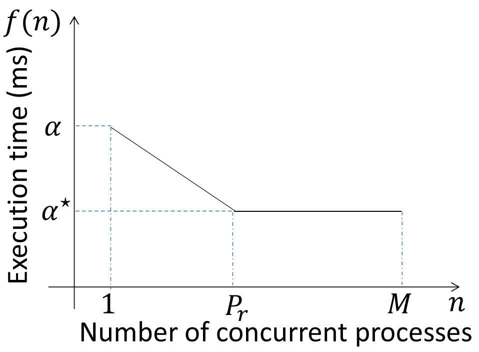
Slow-down policy. With this policy, performance improvement of a service is throttled when number of concurrent requests exceeds a specified value (slow-down point P_s). This policy may due to the parallel execution limitation of the service's implementation. The performance improvement pattern yielded by this policy is depicted in Figure 3.7a. The execution time of the service is given by Equation 3.1.

$$f(n) = \begin{cases} \alpha - \frac{\alpha-\alpha^{\star}}{P_s-1}(n-1), & \text{if } 1 \leq n < P_s \\ \alpha^{\star} - \frac{\alpha^{\star}-\alpha'}{N-P_s}(n-P_s), & \text{if } P_s \leq n \leq N \end{cases} \tag{3.1}$$

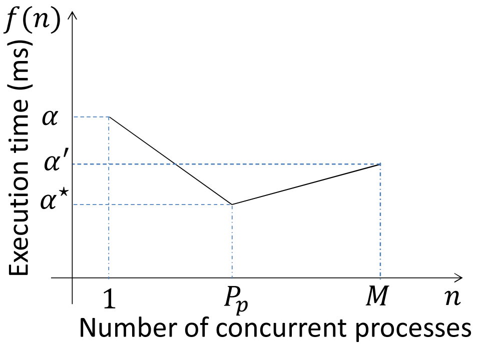
- When $1 \leq n < P_s$, service performance steadily increases with the number of concurrent processes ($\alpha > \alpha^{\star} > \alpha'$).



(a) Slow-down policy



(b) Restriction policy



(c) Penalty policy

Figure 3.7: Performance patterns for parallel execution policies

- When $P_s \leq n \leq N$, service performance continues to improve but at a slower rate. ($\frac{\alpha-\alpha^\star}{P_s-1} > \frac{\alpha^\star-\alpha'}{N-P_s}$).

Restriction policy. Service providers limit the maximum number of concurrent requests that their services can serve. Service performance statures at a specified number of concurrent requests (restriction point P_r). This may be due to limitation of computing resources. This policy creates the performance pattern show in Figure 3.7b. In this case, the execution time of the service is given by Equation 3.2.

$$f(n) = \begin{cases} \alpha - \frac{\alpha-\alpha^\star}{P_r-1}(n-1), & \text{if } 1 \leq n < P_r \\ \alpha^\star, & \text{if } P_r \leq n \leq N \end{cases} \tag{3.2}$$

- When $1 \leq n < P_r$, service performance steadily increases with the number of concurrent processes ($\alpha^\star < \alpha$).
- When $P_r \leq n \leq N$, execution time of the service remains the same ($\alpha' = \alpha^\star$).

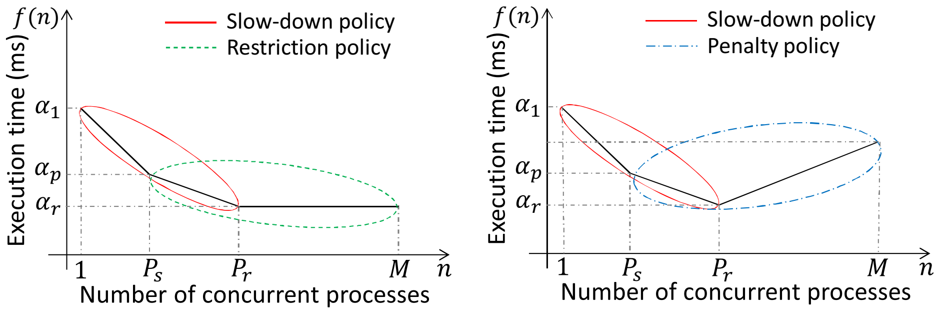
Penalty policy. In some cases, due to limitation of computing resources, or some commercial strategies or security concern, service providers limit parallel execution of their service to a certain number of concurrent requests where the services attain the optimal performance. If the number of concurrent requests sent to the service exceeds the specified number (penalty point P_p), service performance is reduced. The performance improvement pattern of this policy is shown in Figure 3.7c. The execution time of the service is calculated by the Equation 3.3.

$$f(n) = \begin{cases} \alpha - \frac{\alpha-\alpha^\star}{P_p-1}(n-1), & \text{if } 1 \leq n < P_p \\ \alpha^\star + \frac{\alpha'-\alpha^\star}{N-P_p}(n-P_p), & \text{if } P_p \leq n \leq N \end{cases} \tag{3.3}$$

- When $1 \leq n < P_p$, service performance steadily increase with the number of concurrent processes ($\alpha > \alpha^\star$).
- When $P_p \leq n \leq N$, service performance falls ($\alpha' > \alpha^\star$).

Determining value of P

One of the most important things in the proposed policy model is to determine value of P. In this work we assume that at a period of time due to



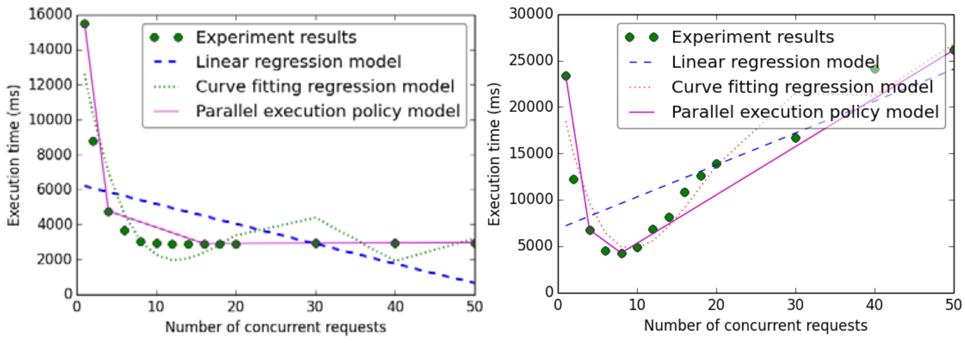
(a) Slow-down and restriction (b) Slow-down and penalty

Figure 3.8: Combination of policies

a specific intention a service provider sets a static value of P for a service. P may change at other time when the provider's intention or conditions are changed. Because the value of P is not explicitly stated, in order to determine P we analyze the observed performance improvement pattern of the service and choose the point that the improvement trend is changed. Since the tested data of the performance improvement is not perfectly linear, in order to find the point where the performance improvement line changes, we use a threshold to determine whether a point is considered to be on the linear line. If the difference between the actual point and the point calculate by using the linear point is smaller the threshold, this actual point is considered as a point on the linear line.

Combination of policies.

From our observation, most of the services employ a combination of different parallel execution policy. We observed two combinations, the first one is combination of slow-down policy and restriction policy as shown in Figure 3.8a. Examples of this combination include J-Server translation service and



(a) J-Server translation service (b) Google translation service

Figure 3.9: Evaluating policy model of different services

Mecab morphological analysis service. The second one is combination of slow-down policy and penalty policy depicted in Figure 3.8b. Examples of this combination include Google translation service and Yandex translation service.

3.6 Evaluation

In this section, we evaluate our proposed policy model in the aspect of how well the model captures performance improvement of web services under parallel execution. Our model is compared with two regression models: a linear fitting model and a curve fitting model with a quartic regression (curve fitting function: $y = ax^4 + bx^3 + cx^2 + dx + e$). Figure 3.9 shows comparison of our policy model and the two regression models of two services: J-Server translation service and Google translation service. The figure shows prediction of the policy model is closer to actual experiment results compare to the other two regression models.

Table 3.1: Comparison of the proposed model with regression models

	S (milliseconds)			R-Squared (%)			P-value
	Linear model	Quartic model	Policy Model	Linear model	Quartic model	Policy model	Policy model
J-Server Tran.	3287.94	1583.47	1049.75	21.3	86.3	92.0	1.23e-09
Google Tran.	5877.13	3035	2460.25	43.8	88.77	90.17	1.31e-08
Mecab	3310.78	1634.90	764.73	19.9	85.3	95.7	3.71e-11
Amazon S3	13734.94	6264.98	4795.82	31.2	89.3	91.6	1.57e-09
Google URL	2080.93	1014.05	698.97	23.2	86.3	91.3	4.06e-09
Tree Tagger	3078.94	1297.93	659.91	1.2	86.8	95.5	5.82e-11
LSD	2521.01	1267.16	885.72	4.5	89.5	93.2	1.56e-09

In more detail, we use standard error (S), and R-squared (R^2) to compare the models. S gives some idea of how much the model's prediction differs from the actual results. R^2 provides an index of the closeness of the actual results to the prediction. S and R^2 are calculated by the following equations:

$$S = \sqrt{\frac{\sum_1^n (Actual_i - Prediction_i)^2}{n-p}}, \text{ and } R^2 = 1 - \frac{\sum_1^n (Actual_i - Prediction_i)^2}{\sum_1^n (Actual_i - mean(Actual))^2}$$

where n is number of observations, p is the number of regression parameters ($p = 2$ in the case of linear regression and our model, $p = 5$ in the case of the quartic regression model).

We also use F-test to calculate P-value for evaluating statistical significance of our proposed model. Table 3.1 shows comparison of the policy model with the two regression models for different web services. The results show that, in all cases, the policy model has the lower standard error and higher R-Squared than either the linear regression model or the quartic regression model. The P-value of the policy model is significantly low (much less than 0.05). We conclude that our policy model has much better accuracy in capturing performance improvement behaviors of web services than the conventional regression models. The policy model is also highly statistically significant and can faithfully estimate the parallel execution effects of web services.

3.7 Conclusion

Parallel execution is well studied in parallel computing and high performance computing. Many models have been proposed to modelling the effect of different factors to the parallel execution efficiency. Most of approaches focused on the side of task/service providers in a sense that service providers can control those factors to increase the parallel execution efficiency. In this work, we focus on the side of users who do not have control on computing resources, implementation of tasks/services. From the view of service users, the limitation of parallel execution of a service is regarded as a policy of the service provider. We introduced a new factor, which is service' policy, that affects parallel execution efficiency of the service.

In this chapter we have analysed performance improvement behaviors of different web services under parallel execution. We provided analyses and evaluations of our parallel execution policy model which includes three types of policies: Slow-down policy, Restriction policy, and Penalty policy. By conducting a series experiments on more than 50 web services, we have experimentally confirmed our model well captures the effects of parallel execution policy. Our model has been proved to be superior to regression models in capturing the parallel execution effects of web services. The evaluation results also showed that our parallel execution policy model can well illustrate performance improvement behaviors of web services under parallel execution. More importantly, our propose model requires less data in order to correctly predict execution time of a web service. However, the three types of parallel execution policies may not correctly cover all types of web service policies. This leaves room for our future work to continue our analysis with larger number of web services to find more types of policies of web services.

Chapter 4

Predicting Parallel Execution Performance of Composite Services

4.1 Introduction

Composing and optimizing a composite service, or a workflow of atomic services, has gained increasing attention in SOA. With the increasing volume of data to be analysed, there is a need of using parallel execution for composite services. Technologies such as data-intensive and many-task computing [Raicu et al., 2012, Humbetov, 2012], and scientific workflows [Taylor et al., 2014] have the potential to enable rapid data analysis for workflows. Many studies have proposed parallel and pipelined execution technique to speed-up workflows [Pautasso and Alonso, 2006], and

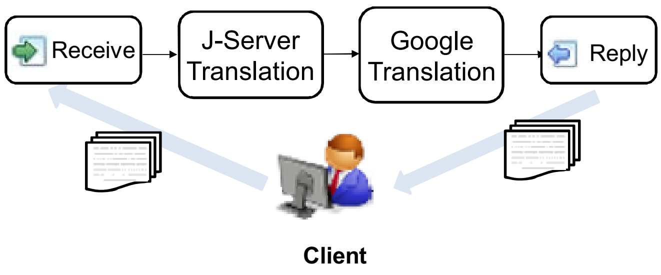


Figure 4.1: A simple composite service

adaptive parallel execution approach for workflows in cloud environments regarding the availability of resources [Oliveira et al., 2012]. These work-s focus on proposing method to support parallel execution for workflows. They do not consider how to optimize the workflow by satisfying requirements of service providers for web services in the workflow. There are also many methodologies focusing on crating composite services which can attain the optimal Quality of Service (QoS) based on linear integer programming [Cardoso et al., 2004, Zeng et al., 2004] or Genetic Algorithms (GAs) [Canfora et al., 2008]. Most of the proposed approaches did not consider atomic services' policies in optimizing the composite services. An unique point in SOA is, a composite service is combination of different atomic services provided by various providers. Different providers may have different decision on supporting parallel execution for their provided services. To raise the efficiency of parallel execution, we need to consider the different policies when configuring degree of parallelism (DOP) for the composite service. Different from current approaches, in this work we focus on predicting the optimal DOP of a composite service by considering the service policies of all participating providers.

Figure 4.1 shows a simple example of composite service. This is a two-

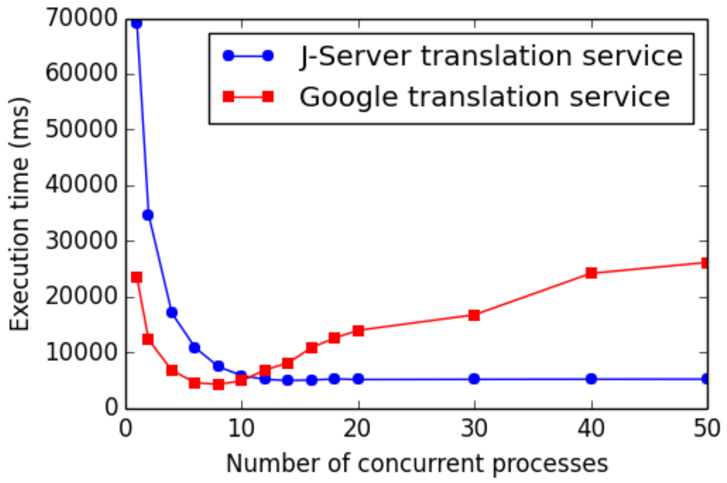


Figure 4.2: Different policies of J-Server and Google translation services

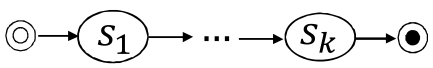
hop translation service, combining J-Server translation service and Google translation service, for translating a document from Japanese to Vietnamese via English. J-Server translation service translates document from Japanese to English, and then Google translation service translates the intermediate translated document from English to Japanese. To reduce execution time, the client invokes the composite service with parallel execution. The input data is split into independent partitions, and several are processed in parallel. By using pipelined execution, if the client configures to process n partitions in parallel, each atomic service will serve n concurrent requests at all the time. From previous chapter, we observed that the two atomic services have different parallel execution policies as shown in Figure 4.2. In this scenari-o, when configuring parallel execution of the composite service we need to specify a suitable DOP of the composite service. This configuration should conform to all atomic services' policies in order to attain the optimal perfor-mance improvement for the composite service. To tackle this problem, in this chapter, we propose a model that embeds parallel execution policies of atomic services into formulae to calculate composite service performance. From the calculation, we predict the optimal DOP for the composite service. Extensive experiments are conducted on real-world translation services. We use several measures such as mean prediction error (MPE), mean absolute deviation (MAD) and tracking signal (TS) to evaluate our model. The analysis results show that our proposed model has good prediction accuracy with regard to identifying optimal DOPs for composite services.

4.2 Predicting Parallel Execution Performance of Composite Services

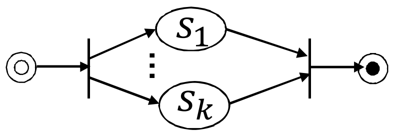
In this section we propose a model that can predict parallel execution performance of composite services. Our model considers policies of all atomic services and the workflow structures for the prediction.

4.2.1 Composite Service Performance

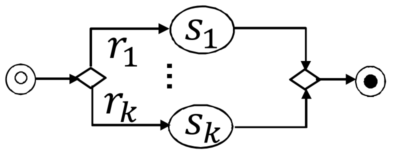
A composite service can be seen as a set of atomic services that cooperate to execute a process that defines the interaction workflow. There are four basic composite structures: *Sequential*, *Parallel*, *Conditional* and *Loop*, see Figure 4.3, where circles represent atomic services and arrows represent the transfer of data between services. QoS of a composite service is aggregate QoS of all atomic services. Existing QoS calculation methods can be classified into two categories: Reduction method with sin-



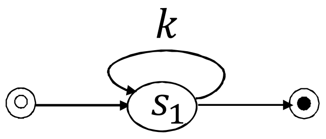
(a) The sequential structure



(b) The parallel structure



(c) The conditional structure



(d) The loop structure

Figure 4.3: Four types of composite structures

gle QoS for service composition [Cardoso et al., 2004, Jaeger et al., 2004], and direct aggregation method with multiple QoSs for the service composition [Ardagna and Pernici, 2007, Yu and Bouguettaya, 2008]. In [Cardoso et al., 2004] several aggregation formulae were proposed to estimate execution time of composite services with different workflow structures. Given that composite service C consists of k atomic services, the formulae to calculate execution time of C is as follows:

- $C = \{s_1, s_2, ..., s_k\}$
- $T(s_i)$ is the execution time of service s_i.
- Aggregate functions to calculate execution time of C for different structures are shown in Table 4.1. In the conditional structure, r_i denotes the probability of service s_i invocation, $\sum_{i=1}^{m} p_i = 1$. In the loop structure, k represents number of iteration of service s_1.

Table 4.1: Aggregation functions to calculate execution time

Structure	Sequential	Parallel	Conditional	Loop
Expected Execution Time	$\sum_{i=1}^{k} T(s_i)$	$\max_{i=1}^{k} T(s_i)$	$\sum_{i=1}^{k} p_i T(s_i)$	$k \times T(s_1)$

We adapt these formulae to calculate execution time of composite services under parallel execution. The parallelism techniques, which are used to improve composite service performance, are described in next section.

4.2.2 Parallel Execution of a Composite Service

As mentioned in chapter 2, in this work, we focus on two parallelism techniques: data parallelism and workflow pipeline execution. We describe more detail about these two techniques in this section.

Data parallelism. When considering data-intensive applications, several input data sets are to be processed using a given workflow. Benefit from the large number of resources available in a grid, workflow services can be instantiated as several computing tasks running on different hardware resources and processing different input data in parallel. Similar to Data Parallelism in atomic service, when a composite service processes large amounts of data sets, the data sets are split into independent partitions, and several computing tasks of each atomic service in the composite service are instantiated to process several partitions in parallel.

Workflow pipeline execution. When a single workflow is operate in parallel on many data partitions. *Workflow pipeline execution* denotes that the processing of several independent partitions by several instances of an atom-

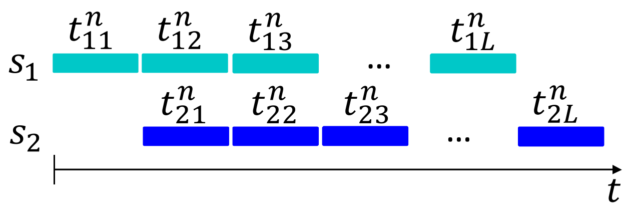


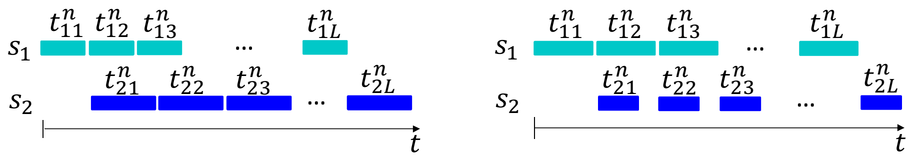
Figure 4.4: Pipeline processing time-line of a composite service

ic service are independent. This parallelism enables pipeline processing of a workflow. That is, when n concurrent requests are sent to a composite service, multiple instances of each atomic service are created and processed in parallel. A pooling technique is used such that when processing M data sets, n out of M data sets are streamed to the composite service in parallel without waiting for responses. The execution of the composite service is done in pipeline manner. Consider an example of a sequential composition of two services. This example yields the pipeline processing time-line shown in Figure 4.4, where $L = \lceil M/n \rceil$ is number of time-steps needed to send M data sets. At the beginning of time period, n data set are sent in parallel. t_{ij}^n is the time that n concurrent processes of service s_i take to finish processing n data sets at time step j. Processing time-line of different control structures are different, and so performance prediction of different structures are also different. We describe prediction for each structure as following.

4.2.3 Performance Prediction of Different Workflow Structures

Sequential Structure

Consider a composite service consisting of two services s_1 and s_2 with *Se-*


t^n_{11} t^n_{12} t^n_{13} ... t^n_{1L}
s_1
t^n_{21} t^n_{22} t^n_{23} ... t^n_{2L}
s_2
t

(a) When $f_1(n) < f_2(n)$ (b) When $f_1(n) > f_2(n)$

Figure 4.5: Processing time-line of sequential structure

quential structure as shown in Figure 4.3a. s_1 and s_2 may have different policies when using parallel execution as specified in Chapter 3. We need to estimate performance of the composite service considering policy of each atomic service.

First, we describe how the parallel execution is applied to this structure. The input data is separated into M partitions. There is a pool to control the number of concurrent partitions sent to the composite service (DOP). Suppose, the pool is configured to send n concurrent partitions to the composite service. Therefore, all M partitions are sent to the composite service in $L = \lceil M/n \rceil$ times. Each atomic service is executed in parallel with n concurrent processes. Let (α_1, $\alpha_1^\star$, α_1', P_1) and (α_2, $\alpha_2^\star$, α_2', P_2) be the parallel execution policies of s_1 and s_2. Execution time of service s_1 and service s_2 to process M partitions, $f_1(n)$ and $f_2(n)$, can be predicted by using our policy model as described in Section 3.5. The pool sends data partitions continuously to the service without waiting for responses. Processing time-line of the composite service is illustrated in Figure 4.5. t^n_{1j} is execution time of n concurrent instances of service s_1 process n partitions at time-step j. t^n_{2j} is execution time of n concurrent instances of service s_2 process n partitions at time-step j. The relative performance of s_1 and s_2 also impacts composite service performance. We consider here two cases:

- ***Case 1:*** Service s_1 has better performance than service s_2 - $f_1(n) < f_2(n)$. The resulting processing time-line is depicted in Figure 4.5a. Let $f_c(n)$ be the execution time of the composite service. In this case $f_c(n)$ is calculated as follows:

  $$f_c(n) = f_2(n) + t_{11}^n \tag{4.1}$$

  t_{11}^n is the time taken by service s_1 to finish processing n partitions in parallel. Suppose that the execution time of s_1 to process each partition is approximate equal. We have:

  $$t_{11}^n \approxeq \frac{f_1(n)}{\lceil M/n \rceil} \tag{4.2}$$

  From Equation (4.1) and (4.2) we have:

  $$f_c(n) \approxeq f_2(n) + \frac{f_1(n)}{\lceil M/n \rceil} \tag{4.3}$$

- ***Case 2:*** service s_2 has better performance than service s_1 - $f_1(n) > f_2(n)$. Processing time-line is depicted in Figure 4.5b. Similarly, $f_c(n)$ is calculated as following equation:

  $$f_c(n) = f_1(n) + t_{2L}^n \approxeq f_1(n) + \frac{f_2(n)}{\lceil M/n \rceil} \tag{4.4}$$

To generalize, let us consider a composite service consisting of a *Sequential* structure of k services $(s_1, s_2, ..., s_k)$. Execution time of each service when processing a large dataset (separated into M partitions) using parallel execution with n concurrent instances are $f_1(n)$, $f_2(n)$, ..., $f_k(n)$ respectively.

Execution time of the composite service ($f_c(n)$) can be predicted as follows:

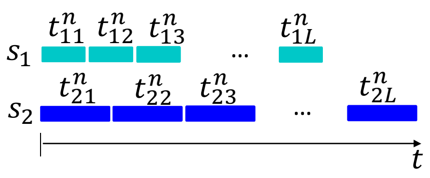
$$
\begin{aligned}
T_{\max} &= \max(f_1(n), f_2(n), ..., f_k(n)) \\
f_c(n) &\approxeq T_{\max} + \frac{\sum_{i=1}^{k} f_i(n) - T_{\max}}{\lceil M/n \rceil}
\end{aligned}
\tag{4.5}
$$

Parallel Structure

Consider a composite service with service s_1 and s_2 in the parallel structure shown in Figure 4.3b. In this case the two services process data in parallel. The original dataset is separated into M partitions, n of which are sent to the composite service concurrently. DOP of the composite service is set to n, so n processes of each atomic service are executed in parallel. Suppose that (α_1, $\alpha_1^\star$, α_1', P_1) and (α_2, $\alpha_2^\star$, α_2', P_2) are parallel execution policies of s_1 and s_2. Execution time of s_1 and s_2, $f_1(n)$ and $f_2(n)$, can be predicted by using the policy model. Similarly, we use a pool to stream data partitions to the service without waiting responses. This yields pipeline execution of the workflow. The processing time-line of the composite service is illustrated in Figure 4.6. t_{1j}^n is execution time of n concurrent instances of service s_1 process n partitions at time-step j. t_{2j}^n is execution time of n concurrent instances of service s_2 process n partitions at time-step j. From this processing time-line, the execution time of the composite service ($f_c(n)$) is predicted as the maximum value of $f_1(n)$ and $f_2(n)$:

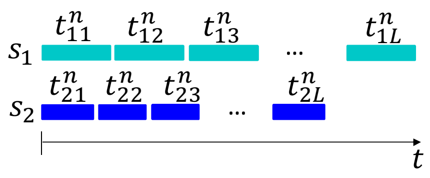
$$
f_c(n) \approxeq \max(f_1(n), f_2(n)) \tag{4.6}
$$

To generalize, consider a parallel combination of k services (s_1, s_2, ..., s_k).



s_1
t_{11}^n t_{12}^n t_{13}^n ... t_{1L}^n
s_2
t_{21}^n t_{22}^n t_{23}^n ... t_{2L}^n
t

(a) When $f_1(n) < f_2(n)$



s_1
t_{11}^n t_{12}^n t_{13}^n ... t_{1L}^n
s_2
t_{21}^n t_{22}^n t_{23}^n ... t_{2L}^n
t

(b) When $f_1(n) > f_2(n)$

Figure 4.6: Processing time-line of parallel structure

Suppose that parallel execution policy of service s_i is (α_i, $\alpha_i^\star$, α_i', P_i), execution time of service s_i with n concurrent processes ($f_i(n)$) can be calculated by using our policy model. In this case, the execution time of the composite service is predicted with the following generalized equation:

$$f_c(n) \approxeq \max_{i=1}^{k} f_i(n) \tag{4.7}$$

Conditional Structure

Figure 4.3c shows a *Conditional* combination of two service s_1 and s_2. Again the composite service processes a big dataset that is separated into M partitions. We configure the pool so that n partitions can be sent to the composite service concurrently. The *Conditional* structure states that a portion of n partitions will be sent to service s_1, the remainder is sent to service s_2. Suppose that the ratios are r_1 and r_2 ($r_1 + r_2 = 1$). This means that $r_1 n$ partitions are processed by s_1 in parallel, while the remaining $r_2 n$ partitions are processed by s_2 concurrently. In total, $r_1 M$ partitions are processed by s_1, and $r_2 M$ partitions are processed by s_2. We configure the parallel execution so that $r_1 n$ concurrent instances of s_1 and $r_2 n$ concurrent instances of s_2 are initiated. Execution time of s_1 and s_2, $f_1(r_1 n)$ and $f_2(r_2 n)$, are calculated as follows:

$$f_1(r_1n) = \begin{cases} \alpha_1 - \frac{\alpha_1-\alpha_1^\star}{P_1-1}(r_1n-1), & \text{if } 1 \le r_1n < P_1 \\ \alpha_1^\star - \frac{\alpha_1^\star-\alpha_1'}{r_1M-P_1}(r_1n-P_1), & \text{if } P_1 \le r_1n \le r_1M \end{cases}$$

$$f_2(r_2n) = \begin{cases} \alpha_2 - \frac{\alpha_2-\alpha_2^\star}{P_2-1}(r_2n-1), & \text{if } 1 \le r_2n < P_2 \\ \alpha_2^\star - \frac{\alpha_2^\star-\alpha_2'}{r_2M-P_2}(r_2n-P_2), & \text{if } P_2 \le r_2n \le r_2M \end{cases}$$

Where (α_1, $\alpha_1^\star$, α_1', P_1) is parallel execution policies of s_1 when processing r_1M partitions and (α_2, $\alpha_2^\star$, α_2', P_2) is that of s_2 when processing r_2M partitions.

When $n = 1$, data partitions are sent to the composite service one at a time, one partition is processed by s_1 or s_2 at a time. The execution time the composite service to process M partitions is execution time of s_1 to process r_1M partitions adds execution time of s_2 to process r_2M partitions. When $n > 1$, since there are r_1n concurrent processes of s_1 and r_2n concurrent processes of s_2, the execution of the conditional structure is similar with parallel structure where s_1 processes rM partitions and s_2 processes r_2M partitions concurrently. In this case, processing time-line of the conditional structure is as illustrated in Figure 4.7; the number of time-steps, $L = \lceil M/n \rceil$. The execution time of the conditional structure $f_c(n)$ is predicted as follows:

$$f_c(n) \approxeq \begin{cases} f_1(1) + f_2(1), & \text{if } n = 1 \\ \max(f_1(r_1n), f_2(r_2n)), & \text{if } n > 1 \end{cases} \tag{4.8}$$

To generalize, consider a *Conditional* structure of k services (s_1, s_2, ..., s_k). Suppose that r_1, r_2, ..., r_k are the ratios of requests sent to each service. We

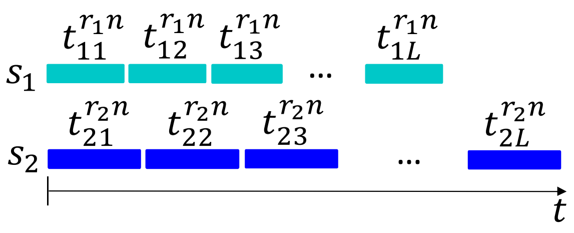


Figure 4.7: Processing time-line of conditional structure

have:

- $\sum_{i=1}^{k} r_i = 1$
- $r_i M$ partitions are processed by s_i
- With n partitions sent to the composite service in parallel, $r_i n$ instances of service s_i will be initiated and processed concurrently.
- Execution time of service s_i when processing $r_i M$ partitions with $r_i n$ concurrent instances can be calculated with $f_i(r_i n)$, where the parallel execution policy of s_i is $(\alpha_i, \alpha_i^{\star}, \alpha_i', P_i)$.

The execution time of the composite service ($f_c(n)$) is predicted by the following equation:

$$f_c(n) \approxeq \begin{cases} \sum_{1}^{k} f_i(1), & \text{if } n = 1 \\ \max_{i=1}^{k} f_i(r_i n), & \text{if } n > 1 \end{cases} \tag{4.9}$$

Loop Structure

Consider the *Loop* structure of a service s_1 as shown in Figure 4.3d. Consider the example of the iteration number of two. In this case, this *Loop* structure can be converted to a *Sequential* structure of two services s_1. The composite service processes M partitions. n processes of service s_1 are executed to process n partitions in parallel. (α_1, $\alpha_1^\star$, α_1', P_1) is parallel execution policy of s_1, execution time of the service is $f_1(n)$ can be predicted by using the policy model. The processing time-line of the composite service is illustrated in Figure 4.8; the number of time-steps is $L = \lceil M/n \rceil$. At the first time step, n requests are sent to s_1 in parallel, t_{11}^n is the time taken by n instances of s_1 to process the first n partitions. From the second time step to $(\lceil M/n \rceil)$ time step, $2n$ requests are sent to service s_1 concurrently, so the time to process n partitions is maximum value of $f_1(n)$ and $f_1(2n)$. Let's ΔT is duration from second time step to $(\lceil M/n \rceil)$ time step, ΔT is calculated as follows:

$$\Delta T \approxeq (\lceil M/n \rceil - 1)\frac{\max(f_1(n), f_1(2n))}{\lceil M/n \rceil}$$

At the last time step, n last partitions are processed in parallel by n concurrent instances of s_1. The execution time of the composite service ($f_c(n)$) is predicted by the equation bellow:

$$\begin{aligned} f_c(n) &\approxeq 2\frac{f_1(n)}{\lceil M/n \rceil} + \Delta T \\ &\approxeq 2\frac{f_1(n)}{\lceil M/n \rceil} + (\lceil M/n \rceil - 1)\frac{\max(f_1(n), f_1(2n))}{\lceil M/n \rceil} \end{aligned} \tag{4.10}$$

To generalize, consider a *Loop* structure where s_1 is executed k time. The structure can be converted to a sequence structure of k service s_1. Processing of the composite service follows pipeline processing time-line. At time-step i ($1 \leq i < k$), in concurrent request are sent to s. From time-step k to $\lceil M/n \rceil$, kn concurrent requests are sent to service s. The execution time of

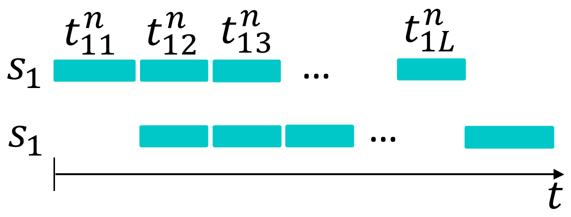


Figure 4.8: Processing time-line of loop structure

the composite service can be calculated by following equation:

$$f_c(n) \approxeq \frac{2\sum_{j=1}^{k-1} \max_{i=1}^{j} f(in)}{\lceil M/n \rceil} + (\lceil M/n \rceil - k + 1)\frac{\max_{i=1}^{k} f(in)}{\lceil M/n \rceil} \tag{4.11}$$

4.2.4 Performance Prediction of Complex Cases

Our prediction model can also deal with complex cases such as combination of different parallel execution policies for one atomic service, or a composite service which contains different control structures.

Combination of Policies

A service provider may employ more than one parallel execution policy for an atomic service. For instance, an atomic service may have both slow-down policy and restriction policy. In this case, to calculate execution time of the service we need to determine each interval of number of concurrent processes in which the service follows only one policy. Then, apply different policy models in different intervals to calculate execution time.

Figure 4.9 shows an example of a combination of the three policies: slow-down policy, restriction policy, and penalty policy:

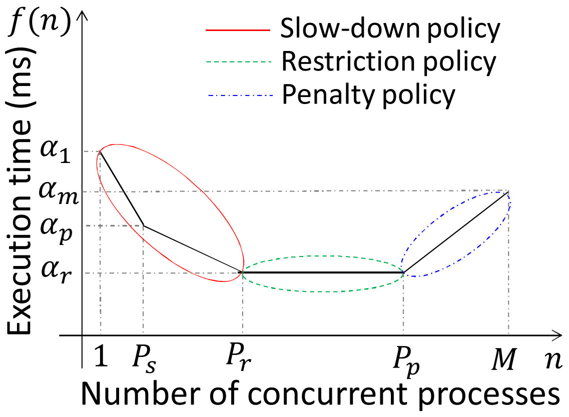


Figure 4.9: Combination of policies

- When $1 \leq n \leq P_r$, the service follows slow-down policy, execution time is calculated by applying Equation 3.1:

$$f(n) = \begin{cases} \alpha_1 - \frac{\alpha_1 - \alpha_p}{P_s - 1}(n-1), & \text{if } 1 \leq n < P_s \\ \alpha_p - \frac{\alpha_p - \alpha_r}{P_r - P_s}(n - P_s), & \text{if } P_s \leq n \leq P_r \end{cases}$$

- When $P_r \leq n \leq P_p$, the service follows restriction policy. Apply Equation 3.2, execution time of the service is: $f(n) = \alpha_r$.
- When $P_p \leq n \leq M$, the service follows penalty policy, execution time is calculated as follows:

$$f(n) = \alpha_r + \frac{\alpha_m - \alpha_r}{M - P_p}(n - P_p)$$

Combine all these intervals, we can calculate execution time of the service by the following equation:

$$f(n) = \begin{cases} \alpha_1 - \frac{\alpha_1 - \alpha_p}{P_s - 1}(n-1), & \text{if } 1 \leq n < P_s \\ \alpha_p - \frac{\alpha_p - \alpha_r}{P_r - P_s}(n - P_s), & \text{if } P_s \leq n < P_r \\ \alpha_r, & \text{if } P_r \leq n < P_p \\ \alpha_r + \frac{\alpha_m - \alpha_r}{M - P_p}(n - P_p), & \text{if } P_p \leq n \leq M \end{cases}$$

Combination of control structures

A composite service may contain different control structures. To calculate execution time of this type of composite service, we first reduce the complex workflow to a simple workflow which contains only sequential structure. After the reduction, it is easily to calculate the execution time of the composite service by applying the equation for the sequential structure.

We adopt the reduction methodology proposed in [Cardoso et al., 2004] to calculate execution time of a complex workflow under parallel execution. The reduction for each structure is as follows:

- *Reduction of parallel structure.*

  Figure 4.10 illustrates how a parallel structure of m services s_1, s_2, ..., s_m can be reduced to a single service s_{1m}. In this reduction, execution time of s_a and s_b remain unchanged. We apply prediction model for parallel workflow as shown in Section 4.2.3 to compute execution time of the reduction. Execution time of s_{1m} is calculate by the Equation 4.7.

- *Reduction of conditional structure.*

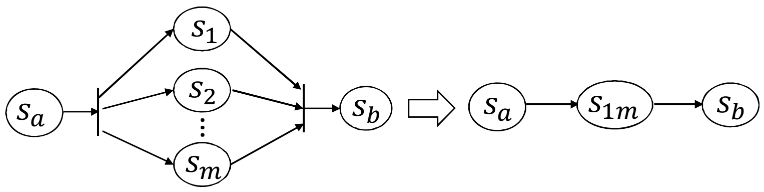


Figure 4.10: Parallel structure reduction

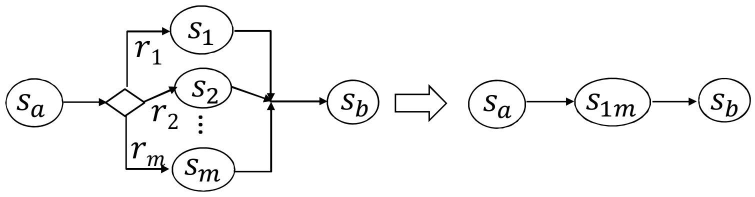


Figure 4.11: Conditional structure reduction

Figure 4.11 illustrates how a conditional structure of m services s_1, s_2, ..., s_m can be reduced to a single service s_{1m}. In this reduction, execution time of s_a and s_b remain unchanged. We apply prediction model for conditional workflow as shown in Section 4.2.3 to compute execution time of the reduction. Execution time of s_{1m} is calculate by the Equation 4.9.

- *Reduction of loop structure.*

  Figure 4.12 illustrates how a loop structure of service s_1 with k iterations can be reduced to a single service s_{1loop}. In this reduction, execution time of s_a and s_b remain unchanged. We apply prediction model for loop workflow as shown in Section 4.2.3 to compute execution time of the reduction. Execution time of s_{1loop} is calculate by the Equation 4.11.

- *Reduction of a complex workflow.*

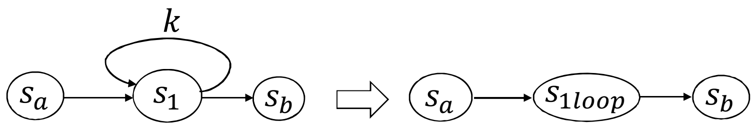


Figure 4.12: Loop structure reduction

Figure 4.13 shows a reduction of a complex workflow with 4 different structures:

- Parallel combination of s_1 and s_2 is reduced to s_{12} part:

$$f_{12}(n) = max(f_1(n), f_2(n)).$$

- Assume that the ratios of request sent to services s_3 and s_4 are r_3 and r_4 ($r_3 + r_4 = 1$). Conditional combination of s_3 and s_4 is reduced to s_{34} part:

$$f_{34}(n) \approxeq \begin{cases} f_3(1) + f_4(1), & \text{if } n = 1 \\ \max(f_3(r_3 n), f_4(r_4 n)), & \text{if } n > 1 \end{cases}$$

- Assume that s_5 is looped with 2 iterations. This loop is reduced to s_{5loop} part:

$$f_{5loop}(n) \approxeq 2\frac{f_5(n)}{\lceil M/n \rceil} + (\lceil M/n \rceil - 1)\frac{\max(f_5(n), f_5(2n))}{\lceil M/n \rceil}$$

The complex workflow is reduced to sequential combination of three part s_{12}, s_{34}, and s_{5loop}, we apply the model for sequential structure to

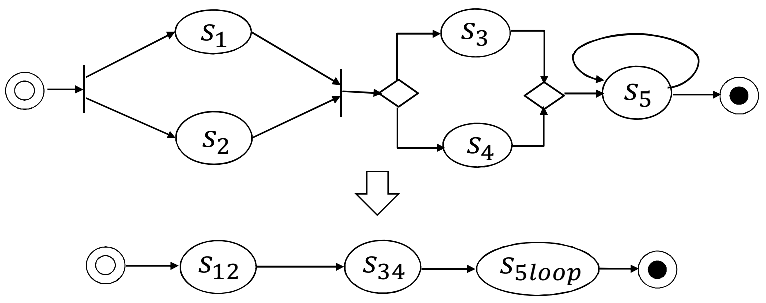


Figure 4.13: Reduction of a complex workflow

calculate execution time of the composite service:

$$T_{\max} = \max(f_{12}(n), f_{34}(n), f_{5loop}(n)),$$
$$f_c(n) \approxeq T_{\max} + \frac{(f_{12}(n) + f_{34}(n) + f_{5loop}(n)) - T_{\max}}{\lceil M/n \rceil}$$

4.2.5 Determining Optimal DOP of a Composite Service

We can obtain the optimum parallel execution by determining the optimal DOP of composite service that gain the optimal performance improvement. Execution time of the composite service ($f_c(n)$) is calculated by using the above model. Different parallel execution policies and different structures yield different $f_c(n)$. In a simple case, we can determine the optimal DOP by estimating the composite service performance on different interval of n. For example, assume that a composite service is a sequential composition of two service s_1 and s_2, where s_1 is covered by the penalty policy specified by (α_1, $\alpha_1^\star$, α_1', P_{p1}), s_2 is covered by the restriction policy specified by (α_2, $\alpha_2^\star$, α_2', P_{r2}). Execution time of s_1 and s_2 are $f_1(n)$ and $f_2(n)$ calculated by using the policy model.

Assume that the relative of $f_1(n)$ and $f_2(n)$ is as follows:

- $\exists P^\star$ where $f_1(P^\star) = f_2(P^\star)$.
- When $n < P^\star$, $f_1(P^\star) < f_2(P^\star)$.
- When $n > P^\star$, $f_1(P^\star) > f_2(P^\star)$.
- $P_{p1} < P^\star < P_{r2}$.

We apply the prediction model for sequential combination to calculate $f_c(n)$ on each interval as follows:

- When $1 < n \leq P_{p1}$ both $f_1(n)$ and $f_2(n)$ decrease when n increases. Since $f_1(n) < f_2(n)$, $f_c(n) = f_2(n) + \frac{f_1(n)}{\lceil M/n \rceil}$. It is obvious that $f_c(n)$ decreases when n increases.
- When $P_{p1} < n \leq P^\star$, $f_1(n)$ increases and $f_2(n)$ decreases when n increases. Since $f_1(n) < f_2(n)$, $f_c(n) = f_2(n) + \frac{f_1(n)}{\lceil M/n \rceil}$. $f_1(n)$ has very small effect to $f_c(n)$, so $f_c(n)$ still decreases when n increases.
- When $P^\star < n \leq P_{r2}$, $f_1(n)$ increases and $f_2(n)$ decreases when n increases. Since $f_1(n) > f_2(n)$, $f_c(n) = f_1(n) + \frac{f_2(n)}{\lceil M/n \rceil}$. $f_1(n)$ has big impact to $f_c(n)$, $f_c(n)$ increases when n increases.
- When $P_{r2} < n \leq M$, $f_1(n)$ increases and $f_2(n)$ remains stable when n increases. Since $f_1(n) > f_2(n)$, by applying Equation 4.3, we have $f_c(n) = f_1(n) + \frac{f_2(n)}{\lceil M/n \rceil}$. It is obvious that $f_c(n)$ increases when n increases.
- In this case, it is easily to estimate that, when $n = P^\star$, $f_c(n)$ gains minimum value.

In general, in order to find optimum value of n where $f_c(n)$ has minimum value. From the function, we determine the linear trend of the equation value in different intervals of n. Finally, we can estimate the optimal point (P_{opt}) where the function has minimum value. The complexity of this process depends on number of atomic services and services' performance relative. If there are many intervals, function $f_c(n)$ is different in each interval, the process becomes very complex and difficult to find the optimum n.

4.3 Evaluation

We conduct experiments to evaluate our prediction model. Specifically, we attempt to answer the following question: How accurate is our prediction model, compared to the actual result?

We created different composite services with different structures and used our proposed model to predict performance of the composite services and estimate the optimal degree of parallelism.

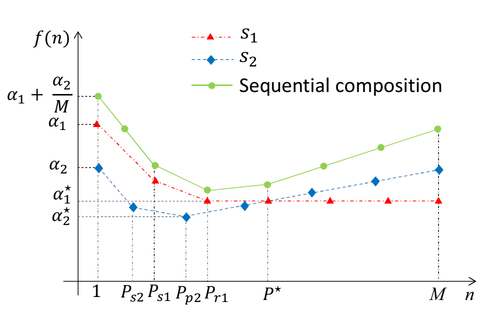
4.3.1 Simple Workflows

Sequential Structure. Consider a case where a composite service has two services s_1 and s_2 in *Sequential* structure with the following conditions:

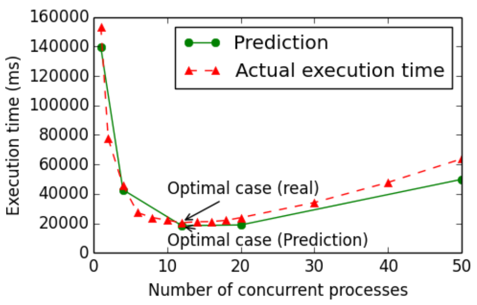
- s_1 employs slow-down and restriction policies specified by P_{s1} and P_{r1}, respectively.
- s_2 employs slow-down and penalty policies specified by P_{s2} and P_{p2}, respectively.

There are several correlation patterns between $f_1(n)$ and $f_2(n)$. Figure 4.14a shows a prediction of a crossed performance case (when the number of concurrent processes is $P^{\star}$: $f_1(P^{\star}) = f_2(P^{\star})$, and $P_{p2} \leq P_{r1} \leq P^{\star}$). The solid line shows the performance pattern of the composite service. In this case, the composite service has optimal performance when $n = P_{r1}$. We explain this prediction as follows:

- When n = 1, $f_1(n) > f_2(n)$, applying Equation 4.3 yields $f_c(1) = \alpha_1 + \frac{\alpha_2}{M}$.
- When $1 < n \leq P_{p2}$, both $f_1(n)$ and $f_2(n)$ decrease when n increases. Since $f_1(n) > f_2(n)$, applying Equation 4.3 yields $f_c(n) = f_1(n) + \frac{f_2(n)}{\lceil M/n \rceil}$. It is obvious that $f_c(n)$ decreases as n increases.
- When $P_{p2} < n \leq P_{r1}$, $f_1(n)$ decreases and $f_2(n)$ increases as n increases. Since $f_1(n) > f_2(n)$, by applying Equation 4.3, $f_c(n) = f_1(n) + \frac{f_2(n)}{\lceil M/n \rceil}$. $f_2(n)$ has very small effect to $f_c(n)$, so $f_c(n)$ still decreases as n increases.
- When $P_{r1} < n \leq P^{\star}$, $f_1(n)$ remains stable and $f_2(n)$ increases when n increases. Since $f_1(n) > f_2(n)$, by applying Equation 4.3, $f_c(n) = f_1(n) + \frac{f_2(n)}{\lceil M/n \rceil}$. We can see that $f_c(n)$ will slightly increase as n increases.
- When $P^{\star} < n \leq M$, $f_1(n)$ remains stable and $f_2(n)$ increases when n increases. Since $f_1(n) < f_2(n)$, applying Equation 4.3 yields $f_c(n) = f_2(n) + \frac{f_1(n)}{\lceil M/n \rceil}$. It is obvious that $f_c(n)$ increases when n increases.
- The composite service attains optimal performance when $n = P_{r1}$.



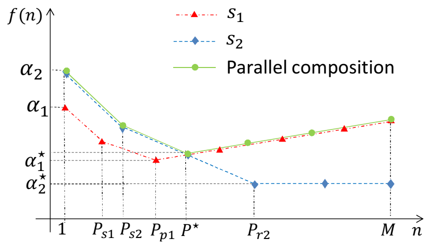
(a) An ideal prediction



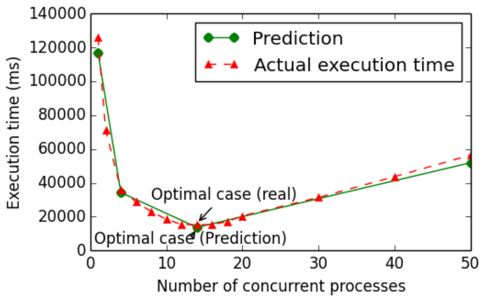
(b) Combination of J-Server and Google translation services

Figure 4.14: Evaluating sequential structure

As a real example for this case, we create a back translation service from two translation services J-Server translation service and Google translation service. J-Server employs slow-down and restriction policies with $P_{sjserver} = 4$ and $P_{rjserver} = 12$. Google translation service employs slow-down and penalty policies with $P_{sgoogle} = 4$ and $P_{pgoogle} = 8$. When number of concurrent processes $n = 20$, $f_{google}(n) = f_{jserver}(n)$. Performance prediction of the composite service when translating a document, containing 500 paragraphs from Japanese to English, is shown in Figure 4.14b. The solid line is the prediction given by our prediction model, while the dash line is the real execution time of the composite service. In this example, the model predicts that when $n = P_{rjserver} = 12$, the composite service attains its best performance. This prediction matches the real result. However, the predicted execution time is not so accurate, the best performance of the composite service calculated by the model was 18384 milliseconds, while the real result was 20317 milliseconds. With the optimal DOP, performance of the composite service improve significantly, i.e., execution time of the composite service decreased to nearly 86 percent.



(a) An ideal prediction



(b) Combination of Google and Bing translation services

Figure 4.15: Evaluating parallel structure

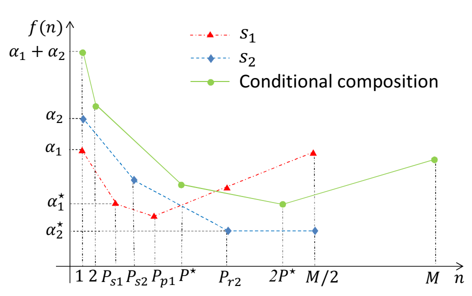
Parallel Structure. We created a composite service from two services s_1 and s_2 with the *Parallel* structure. According to the prediction model, execution time of the composite service, f_c, is calculated as $f_c(n) = \max(f_1(n), f_2(n))$. It is easy to predict the performance of the composite service. With the case in Figure 4.15a, according to the prediction model the composite service reaches its best performance when $n = P^\star$, where $f_1(P^\star) = f_2(P^\star)$.

As a real example of this case, we created a *Parallel* combination of Google translation service and Bing translation service. A document with 500 paragraphs was translated in parallel from Japanese to English. Figure 4.15b compares the prediction (solid line) and the real execution time (dash line) of the composite service. According to the prediction, when $n = 14$ ($f_{google}(14) = f_{bing}(14)$), the composite service attains its best performance; this prediction matches to the real result. However, the prediction of the best execution time of composite service was not so precise, i.e., the model predicted the best execution time of the composite service to be 13642 milliseconds, while the real result was 15181 milliseconds. With the optimal DOP, performance of the service increased by approximately 87 percent.

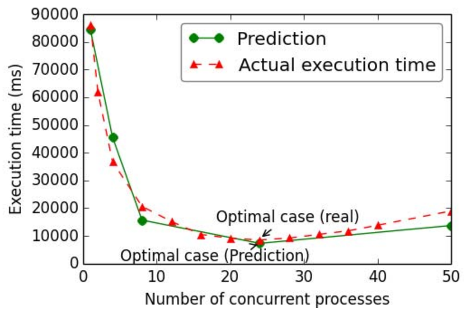
Conditional Structure. Consider the case where a composite service is a *Conditional* composition of two service s_1 and s_2. s_1 employs slow-down and penalty policies, while s_2 employs slow-down and restriction policies. Figure 4.16a shows a performance prediction of the composite service when the ratios of requests sent to each services $r_1 = r_2 = 0.5$. $P^\star$ is number of concurrent processes when $f_1(P^\star) = f_2(P^\star)$. The solid line shows that the composite service has optimal performance when $n = 2P^\star$. We explain this result as follows:

- When n = 1, by appling Equation 4.8, we have $f_c(1) \approxeq f_1(1) + f_2(1) = \alpha_1 + \alpha_2$.
- When $1 < n/2 \leq P^\star$, $f_2(n/2)$ decreases when n increases. Since $f_1(n/2) < f_2(n/2)$, by applying Equation 4.8, we have $f_c(n) \approxeq \max(f_1(n/2), f_2(n/2)) = f_2(n/2)$. It is obvious that $f_c(n)$ decreases when n increases.
- When $P^\star < n/2 \leq M/2$, $f_1(n/2)$ increases when n increases. Since $f_1(n/2) > f_2(n/2)$, by applying Equation 4.8, we have $f_c(n) \approxeq \max(f_1(n/2), f_2(n/2)) = f_1(n/2)$. It is obvious that $f_c(n)$ increases when n increases.
- The composite service gains optimal performance when $n = 2P^\star$.

To give real example for this case, we created a *Conditional* combination of two translation services Google translation service and Baidu translation service. Performance prediction (solid line) and the actual execution time (dash line) of the composite service, when processing a document with 500 paragraphs, are shown in Figure 4.16b for $r_1 = r_2 = 0.5$. The result shows that the prediction model precisely predicts the optimal case. In this exam-



(a) An ideal prediction when $r_1 = r_2 = 0.5$



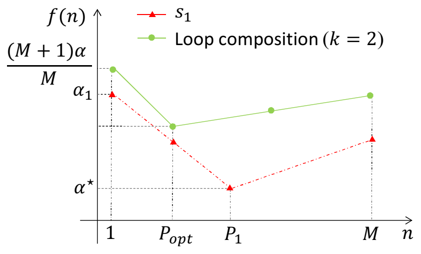
(b) Combination of Google and Baidu translation services

Figure 4.16: Evaluating conditional structure

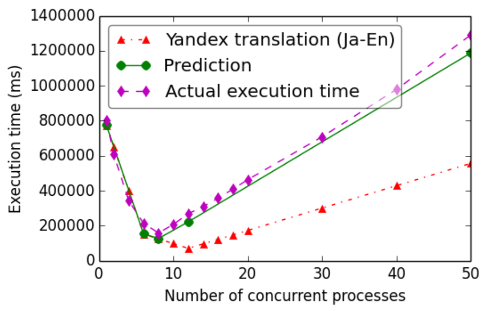
ple, when $n = 24$, the composite service attains the best performance. This matches the actual result. However, the prediction of execution time is not so accurate. The best execution time of the composite service calculated by the model was 7265 milliseconds, while the real result was 8479 milliseconds. In the optimal case, execution time of the composite decreased by nearly 86 percent.

Loop Structure. Consider the case of a composite service that demands service s_1 in a *Loop* structure. In a simple case, the loop iteration is 2. Suppose that the service s_1 combines slow-down and penalty policies specified by P_{s1} and P_{p1}. Performance prediction of the composite service is shown in Figure 4.17a. The solid line shows the predicted performance of the loop composition. In this case the composite service attains the best performance when $n = P_{opt}$, where $f_1(2n)$ starts surpassing $f_1(n)$. We explain this result as follows:

- When n = 1, by applying Equation 4.10, we have $f_c(1) \approxeq 2\frac{f_1(1)}{M} + (M-1)\frac{\max(f_1(1), f_1(2))}{M} = 2\frac{\alpha}{M} + \frac{M-1}{M}\alpha = \frac{M+1}{M}\alpha$.



(a) An ideal prediction

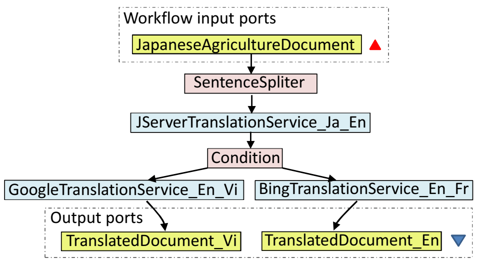


(b) Loop of Yandex translation service (k = 2)

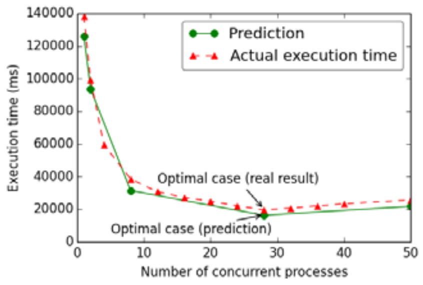
Figure 4.17: Evaluating loop structure

- When $1 < n \leq P_{opt}$, $f_1(2n) < f_1(n)$, by applying Equation 4.10, we have $f_c(n) \approxeq \frac{2f_1(n)}{\lceil M/n \rceil} + \frac{\lceil M/n \rceil - 1}{\lceil M/n \rceil} \max(f_1(n), f_1(2n)) = \frac{\lceil M/n \rceil + 1}{\lceil M/n \rceil} f_1(n)$. Since $f_1(n)$ decreases as n increases, $f_c(n)$ decreases when n increases.
- When $P_{opt} < n \leq M$, $f_1(2n) > f_1(n)$, by applying Equation 4.10, we have $f_c(n) \approxeq \frac{2f_1(n)}{\lceil M/n \rceil} + \frac{\lceil M/n \rceil - 1}{\lceil M/n \rceil} \max(f_1(n), f_1(2n)) = \frac{2f_1(n)}{\lceil M/n \rceil} + \frac{\lceil M/n \rceil - 1}{\lceil M/n \rceil} f_1(2n)$. Since $f_1(2n) > f_1(n)$ and $f_1(2n)$ increases as n increases, so $f_c(n)$ increases as n increases.

As a real example for this case, consider a loop of Yandex translation service with iteration number of 2. This loop is converted into a *Sequential* structure of two Yandex translation services. Figure 4.17b shows performance prediction of the composite service when translating the 500 paragraphs document. The solid line is the prediction of the model, while dash line is the actual execution time of the composite service. The result shows that, our model precisely predicted the optimal DOP of the composite service. In this example, when $n = 8$, the composite service attains its best performance. However, predicted execution time is not so precise. The best


Workflow input ports
JapaneseAgricultureDocument
SentenceSpliter
JServerTranslationService_Ja_En
Condition
GoogleTranslationService_En_Vi
BingTranslationService_En_Fr
Output ports
TranslatedDocument_Vi
TranslatedDocument_En

(a) A complex two-hop translation service



(b) Evaluation of the complex composite service

Figure 4.18: Evaluating a complex composite service

execution time calculated by the model was 126317 milliseconds, while the real execution time was 158038 milliseconds. With the optimal configuration the performance improved approximately 80 percent.

4.3.2 Complex Workflow

We consider here a realistic case when a Japanese agriculture expert who wants to translate a Japanese document that contains two parts, one is information about rice and the other is information about fertilizer. The former is intended to transfer information to Vietnamese farmers, while the latter is for French fertilizer suppliers. We assume that there is no direct translation services from Japanese to Vietnamese and French. The Japanese expert does not want to translate the whole document into Vietnamese or French due to high cost of the translation. In order to do this task we create a composite service shown in Figure 4.18a. This composite service is combination of three translation services with two structures, i.e. *Sequential* structure and *Conditional* structure. First, the document is translated into English using J-Server translation service. Then, that part of translated document, containing information about rice, is translated into Vietnamese by Google translation service. The other part, containing information about fertilizer-s, is translated into French by Bing translation service. J-Server and Bing employ slow-down and restriction policies with $P_{sjserver} = 4$, $P_{rjserver} = 12$ and $P_{sbing} = 4$, $P_{rbing} = 14$, Google employs slow-down and penalty policies with $P_{sgoogle} = 4$ and $P_{pgoogle} = 8$. Figure 4.18b shows a performance prediction when the composite service translates a document of 500 paragraphs (250 paragraphs about rice, 250 paragraphs about fertilizer). The solid line shows the execution time predicted by our model, while the dash line is the actual execution time. Our model predicts that the composite service attains best performance when the number of concurrent processes is 28 (28 concurrent processes of J-Server, 14 concurrent processes of Google, and 14 concurrent processes of Bing) which matches the real result. However predicted execution time is not so precise, the best execution time calculated by the model was 16164 milliseconds, while the real result was 19495 milliseconds. With the optimal DOP, execution time of the composite service decreased by nearly 85%.

Prediction Accuracy

In order to evaluate the accuracy of the proposed model we invoked the above composite service with 15 different agriculture documents with different sizes ranging from 100 paragraphs to 1500 paragraphs. We used the following measures to evaluate accuracy of the proposed model in predicting optimal DOP and optimal execution time.

- First we calculate the difference between actual result and the prediction, this difference indicates errors of the prediction: Error (e) = Actual result - Prediction result.

Table 4.2: Prediction accuracy evaluation

Input data	Optimal Degree of Parallelism (DOP)			Optimal Execution time (millisecond)		
	Prediction	Actual result		Prediction	Actual result	
100 paragraphs	28	24	MPE = 0 MAD = 1.07 TS = 0	2596	3531	MPE = 1204.33 MAD = 1204.33 TS = 15
200 paragraphs	24	24		5373	6550	
300 paragraphs	24	24		8390	9360	
400 paragraphs	24	24		11386	12670	
500 paragraphs	24	28		13984	15287	
600 paragraphs	28	28		17580	18574	
700 paragraphs	24	24		23177	24696	
800 paragraphs	24	24		25273	26344	
900 paragraphs	24	28		30170	31146	
1000 paragraphs	28	28		35567	36627	
1100 paragraphs	24	24		42164	43238	
1200 paragraphs	28	28		43960	45789	
1300 paragraphs	24	24		46757	48043	
1400 paragraphs	28	24		54229	55631	
1500 paragraphs	28	28		57951	59136	

For *n* time periods where we have actual results and prediction values, we calculate:

- Mean Prediction Error (MPE): $MPE = \sum_{i=1}^{n} (e_i)/n$
- Mean Absolute Deviation (MAD): $MAD = \sum_{i=1}^{n} |e_i|/n$
- Tracking signal (TS): $TS = \sum_{i=1}^{n} e_i/MAD$

While MAD is a measure that indicates the absolute size of the errors, the MPE measure indicates the prediction model bias. The ideal value of MPE is 0, when $MPE < 0$ the prediction model tends to over-predict, when $MPE > 0$ the model tends to under-predict. The tracking signal (TS) checks whether there is some bias or not. It simply consists of the summation of the errors over all prediction events. Theoretically, if there is no bias, this sum should remain close to zero. The division by the MAD aims at measuring the distance from the mean in terms of MAD. A prediction model has a good prediction accuracy if the value of TS close to zero. A control limit of TS for a good prediction model is typical in (-4, 4). Table 4.2 shows evaluation of the model in two aspects:

- Predicting the optimal number of concurrent processes:
  - $MPE = 0$ and $MAD = 1.07$. This means that the model yields good predictions; the average absolute error is 1.07 units.
  - TS = 0. This means that in overall there is no bias of the prediction. We can assume that our proposed model well predicts the optimal DOP.
- Prediction of the optimal execution time:
  - $MPE = MAD = 1204.33$. This means that the model tends to under-predict, with an average absolute error of 1204.33 milliseconds.
  - $TS = 15$. This value of TS shows that the model is not so accurate in predicting the optimal execution time.

The proposed prediction model is not so accurate and always under-predict the optimal execution time. One reason for this is that our current model omits some parallel overhead such as time for creating and terminating threads. The accuracy of the model would be improved by adding the parallel overhead time to calculate execution time of an atomic service under parallel execution. We will consider this issue in our future works.

4.4 Conclusion

This this proposed a prediction model that considers the policies of the atomic service providers in predicting the performance of a composite service under parallel execution. To the best of our knowledge, this is the first attempt to incorporate service providers' decisions into parallel computing for service composition. We embedded the proposed policy model to create different formulae, calculating performance of composite services, for different workflow structures. Four workflow structures are considered in the proposed model: Sequential structure, parallel structure, conditional structure and loop structure. Using these formulae we can calculate execution time of composite services when using parallel execution and estimate the optimal degree of parallelism for the composite services. Our model is helpful to build an architecture to control parallel execution of workflows with optimal DOP to attain best performance improvement.

We conducted experiments on real-world translation services to evaluate accuracy of our model. The analysis results show that our model has a good prediction accuracy with regard to identifying optimal degree of parallelism for composite services. Our model is, however, not so accurate in predicting the optimal execution time. Our future work includes improving the model to increase prediction accuracy and extending the model for other QoS metrics such as cost and reputation.

Chapter 5

Implementation of Policy-Aware Parallel Execution Control Architecture

We design an architecture that uses our proposed prediction model to control parallel execution of composite services. This chapter describes some implementation issues in realizing the architecture. We will implement this architecture as an extension to the Language Grid platform[1].

5.1 Introduction

With the maturing of service computing technologies, various programs and data have become available as Web services. In NLP for example, many

[1]The Language Grid: http://langrid.org/en/index.html

language resource providers want to share their resources as language services. The Language Grid [Ishida, 2011] provides an service-oriented platform which enables service providers registering, sharing and combining language services. In the Language Grid, service interfaces are standardized according to the defined service types. This allows us to realize various non-functional requirements by only selecting the appropriate service once a composite service is modeled based on the standardized interfaces of constituent services.

However, Web services are provided by different service providers with a wide variety of policies, such as parallel execution policies as we have introduced in previous chapters. When execute a composite service the system need to consider to satisfy the policies of all service providers concerned, while optimizing QoS. Previous chapters introduced a model regarding parallel execution policies of atomic service to estimate optimal degree of parallelism of a composite service where it attain the optimal performance improvement. In this chapter, we proposed an implementation of the policy-aware parallel execution control architecture which use the proposed model to control parallel execution of composite services.

Our objective is to implemented this architecture as an extended component for the Language Grid. To realize this architecture we face with the following issues:

- The Language Grid introduces Service Workflow Executor to invoke composite services. Current workflow engines, used in this executor, lack of parallel and pipelined execution support. To support parallel and pipelined execution for workflows we need to integrate an pipeline engine to the Service Workflow Executor component.

- The Language Grid support a dynamic binding for concrete services in a workflow at run-time. The optimal degree of parallelism of the workflow changes with different binding services. Therefore, it is necessary to dynamically generate parallel execution deployment for workflow at run-time when concrete service is bound.
- In a multiple users environment such as the Language Grid, a concrete service may be used in multiple workflows. In this case, if those workflows are invoked in parallel, regarding parallel execution policy of the shared service, the architecture should able to allocate suitable parallel processes for each workflow in order to attain optimal execution time for all workflows.

5.2 Design Goal

In this section, we first describe a scenario of controlling parallel execution of a composite service. On a service-oriented collective intelligence platform like the Language Grid, we need to ensure not to violate service providers' policies during execution of a composite service. Then we give an overview of our proposed architecture.

5.2.1 Scenario

Parallel execution control for single workflow

Take a composite service for two-hop translation deployed on the Language Grid as an example. This composite service is used to translate documents

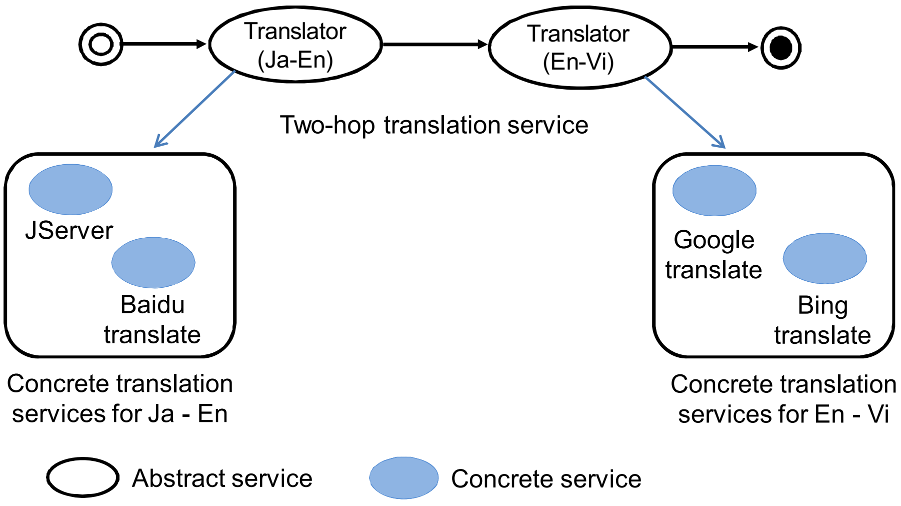


Figure 5.1: A composite service for two-hop translation

with a language pair which is not supported by one translation service. This composite service combines two translation services sequentially to do the translation task. Let take a translation of a document from Japanese and Vietnamese as an example.

Figure 5.1 shows the overview of the composite service. First, the Japanese document is translated from Japanese (Ja) to English (En) using a translation service supporting Ja-En pair. Then, the intermediate translated document is translated from English to Vietnamese (Vi) using a service supporting En-Vi pair. We assume that the composite service is defined in a workflow description language (typically is WS-BPEL). In this composite service description, the constituent services are defined with only the interfaces, and they are not bound to any concrete endpoint. These are called *abstract services*. Endpoint of each abstract service is determined when the composite service is invoked. The service to which an endpoint is bound is called a *concrete service*.

We show the process of execution of this composite service as follows. When user send a request of translating a document, user may also need to specify binding information for concrete services, or in an automatic way, the system will select suitable concrete services for the composite service. Many researches have been proposed in selecting services to compose an optimal composite service. In the context of the Language Grid, Hassine et al. [Hassine et al., 2011] proposed methods to automatically composing composite services based on a constraint optimization problem. After concrete services are bound, the workflow execution engine start to execute the workflow. Suppose that the input document is big and can be split into independent partitions. In order to reduce execution time, the workflow engine need to use parallel execution. Data parallelism and pipeline execution can be used here. The input data is split into multiple partitions, and many partitions are streamed to the workflow in parallel. Each service will have to serve multiple requests in parallel. Due to different parallel execution policies of concrete services, we need to control parallelism of the workflow to attain optimal performance improvement.

To provide such parallel execution control architecture for the Language Grid, we have to deal with several issues. The first issue is to add parallel and pipelined execution support for current workflow engine. The second issues is to generate parallel execution control information (parallel execution deployment) for the workflow, this information is then interpreted by the extended workflow engine.

Parallel execution control for multiple workflows

Suppose that there is another workflow as shown in Figure 5.2. This is also two-hop translation service to translate a half of a document from Japanese

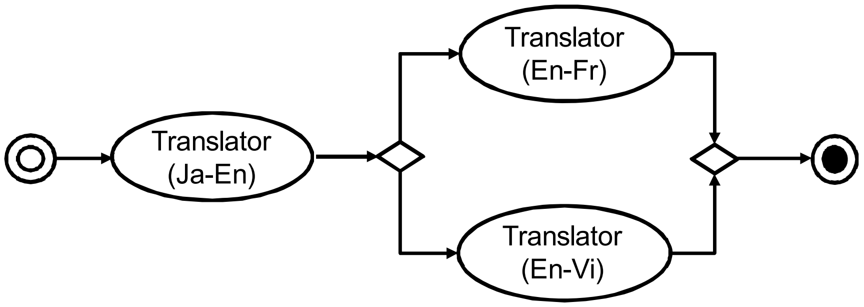


Figure 5.2: A more complex two-hop translation service

(Ja) to French (Fr) and the other half of the document from Japanese to Vietnamese (Vi) via English (En). In multiple users environment, this workflow and the workflow introduced above are invoked in the same time. There is a case that the Translators for En-Vi in both workflows are bound to the same concrete service. Using our proposed prediction model introduced in chapter 4, we can estimate optimal DOP of the two workflows and our architecture will generate parallel execution deployments for each workflow. However, if the two workflow are invoked in parallel, using the generated parallel execution may violate the parallel execution policy of the service bound for the Translator (En-Vi). This may result in the increase of total execution time of two workflows. Therefore, our architecture need to allocate suitable number parallel processes for each workflow in order to attain minimized execution time of both workflows.

5.2.2 Architecture Overview

Here we proposed an architecture to control parallel execution of composite services based on parallel execution policies of atomic services. Figure 5.3 gives an overview of this architecture.

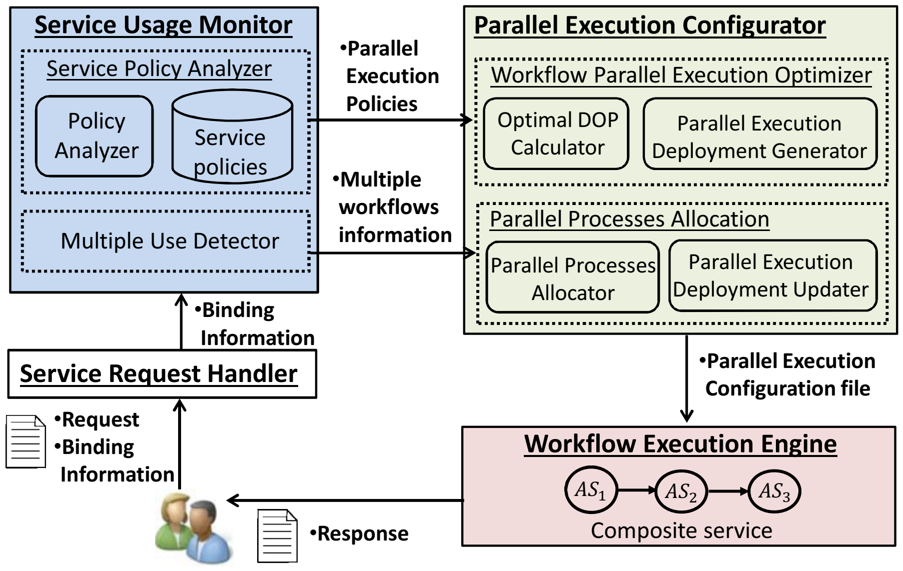

Service Usage Monitor
Service Policy Analyzer
Policy Analyzer
Service policies
Multiple Use Detector
•Parallel Execution Policies
•Multiple workflows information
Parallel Execution Configurator
Workflow Parallel Execution Optimizer
Optimal DOP Calculator
Parallel Execution Deployment Generator
Parallel Processes Allocation
Parallel Processes Allocator
Parallel Execution Deployment Updater
•Binding Information
Service Request Handler
•Parallel Execution Configuration file
•Request
•Binding Information
Workflow Execution Engine
AS_1
AS_2
AS_3
Composite service
•Response

Figure 5.3: Policy-Aware Parallel Execution Control Architecture

The architecture consists of two parts: A service usage monitor and a parallel execution configurator. The service usage monitor gets concrete atomic services endpoint from binding information, these atomic services are analysed and parallel execution policy of each service is determined. This component also monitor multiple use of an atomic service, it detects workflows which have invocation to the same atomic service. Parallel execution policies and multiple workflows information then are passed to the Parallel Execution Configurator. This component calculates optimal DOP of a workflow using our proposed prediction model, and then it generates a parallel execution deployment for the workflow. If there is multiple use of an atomic service from multiple workflows, the configurator will re-calculate suitable parallel processes for each workflow and update the parallel execution development. The workflow execution engine then interprets the parallel execution development and execute the workflows.

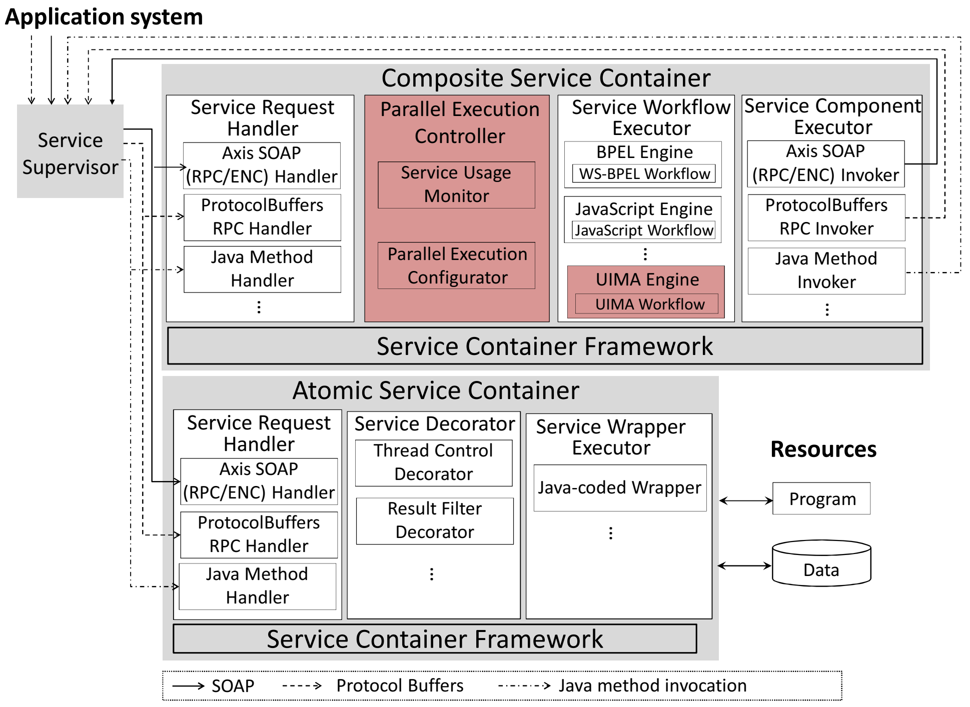


Figure 5.4: Policy-Aware Parallel Execution Control for Language Grid

This architecture can serve as middle-ware for SOA platforms to support and control parallel execution of composite services. Typically, in this work, we implement this architecture as an extended component in the Language Grid Architecture [Murakami et al., 2011]. Figure 5.4 shows the extension of the Language Grid architecture with parallel execution control. We focus on two implementation issues to realize this extension. The first issue is to extend the workflow execution engine to support parallel execution. The second issue represent parallel execution control information to be easily interpreted by the workflow execution engine.

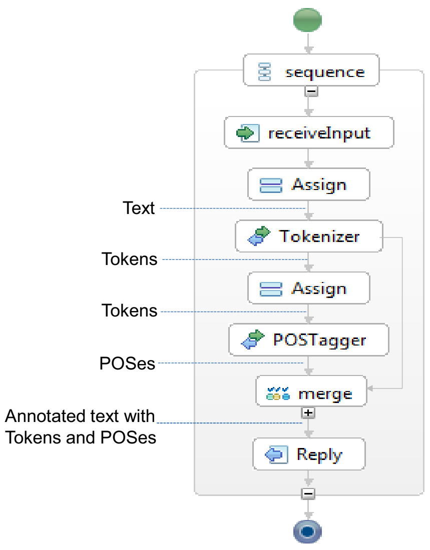
5.3 Parallel Execution Support for Workflow Execution Engine

The Language Grid was built based on SOA. It uses a service workflow engine, such as WS-BPEL execution engine to execute workflows. However, this workflow engine lacks of parallel and pipelined execution support. Processing pipeline based approach is another approach for combining different tasks. This approach focuses on combining language resources in a pipeline to process large-scale data. It has good parallel and pipelined execution support. One typical example is UIMA [Ferrucci and Lally, 2004]. UIMA has become one of the most popular architecture in NLP community, it is core technique for IBM to build the well-known Watson DeepQA system [Ferrucci et al., 2010]. To support parallel and pipeline execution for the Language Grid, we integrated UIMA engine as a workflow execution engine in the Language Grid. We address the following issues to realize this integration.

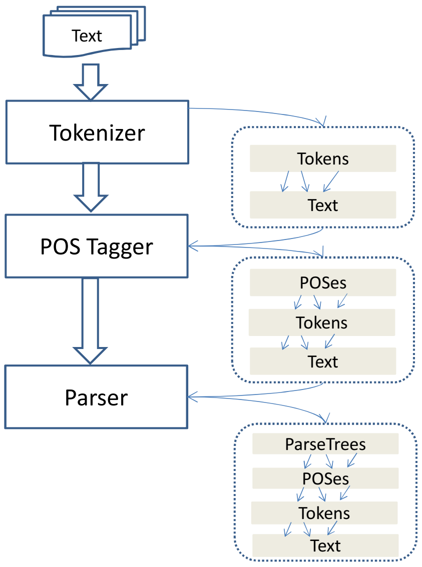
5.3.1 Mapping Service Interface Invocation and Stand-off Annotation

The service workflow approach and processing pipeline approach based on two different models. The former employs service interface invocation model, whereas the latter follows stand-off model. Figure 5.5 shows examples of the these two approaches.

In service workflow approach (Figure 5.5a), each service is defined with



(a) Service workflow



(b) Processing pipeline

Figure 5.5: Service workflow and processing pipeline

an functional interface. For example, a Tokenizer service is defined by a function with input is plain text and output is set of tokens. In order to combined services in a workflow, the services' interface must be compatible, i.e, output of a previous service matches with input of the later service in the workflow. With the stand-off model, on the other hand, in processing pipeline approach, language resources are defined as annotators. In a pipeline, each annotator processes with document and enriches the document with annotation. The annotated document is represent in a Common Data Exchange Format (CDEF) and the CDEF document is passed along components in the pipeline.

CDEF plays an important role in helping the components in a pipeline to work together. Many de-facto standards have been proposed to define CDEF such as in [Vanhoutte, 2004, Ide and Romary, 2009]. In UIMA, CDEF ba-

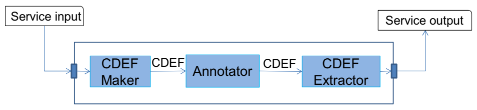
```
<?xml version="1.0" encoding="UTF-8"?>
<annotatedDoc>
   <doc id="1" mimeType="text"
        docString="Text of the document"/>
   <annotations>
      <annot type="POS" docID="1" begin="1"
             end="5" componentID="POSTager">
         <fs>
            <f name="lemma" value="Text"/>
            <f name="postag" value="noun"/>
            ...
         </fs>
      </annot>
      ...
   </annotations>
</annotatedDoc>
```

Figure 5.6: CDEF structure in UIMA

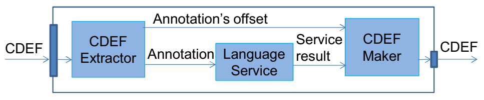
sically consists of two parts: one representing document text, and the other representing annotations.

In order to interwork two kinds of systems, first we need to map between service inter-face and CDEF format. The mapping is defied so as to map input/output of language service to annotation type in CDEF. We define CDEF Maker and CDEF Extractor to conduct the mapping and create two wrappers: Language Service Wrapper and Annotator Wrapper, see Figure 5.7a and Figure 5.7b respectively. The former is used to wrap an annotator as a language service. The latter is used to wrap a language service as an annotator.

- CDEF Extractor manipulates with CDEF to extract annotation and maps it with in-put/output of a service. Annotation type and structure are extracted from CDEF document. The Extractor then maps the annotation with a corresponding object type which is served as input or output of a service.



(a) Language service wrapper



(b) Annotator wrapper

Figure 5.7: Wrappers

- CDEF Maker maps input/output of language services to annotation types and creates CDEF document.

With this mapping, processing pipeline components can be wrapped as services in service workflow framework and can be combined with other services to define composite service. Language service can be wrapped as an annotator and used in pipeline flow.

5.3.2 Adapting Pipeline Engine as Service Workflow Execution Engine

First we adapt the pipeline engine to be able to interpret different structures of workflow such as sequential structure, parallel structure, conditional structure and loop structure by introducing different flow controller to pipeline engine to interpret and execute with these structures: *Sequential flow controller*, *Parallel flow controller*, *Conditional flow controller*, and *Loop flow controller*.

To use the pipeline engine to execute a composite service, we then convert the composite service representation to a pipeline representation while keeping semantic of the composite service. Each atomic service in the composite service is wrapped as an annotator by using the Annotator Wrapper. These annotators are bound to the pipeline representation.

5.4 Parallel Execution Control

5.4.1 Service Usage Monitor

First, the system needs to determine parallel execution policies of atomic services from the binding information. For an atomic service, the Policy Analyzer will check whether the policy of the service is already exist in the database or not. If the policy does not exist, the Policy Analyzer will conduct a test for the service. Based on the policy model, introduced in chapter 3, it determines the policy pattern of the service and stores the policy information into the service policies database.

At the same time, the system monitor multiple use of an atomic service in workflows. From the multiple requests and binding information, the Multiple Use Detector will check whether an atomic service is bound in more than one workflows. The Detector will return all information of workflows that bind to the shared service.

The parallel execution policies of atomic services and the multiple workflows information are passed to the Parallel Execution Controller part.

5.4.2 Parallel Execution Configurator

Workflow Parallel Execution Optimizer

Using our proposed prediction model (chapter 4), the Workflow Parallel Execution Optimizer calculate optimal DOP of each workflow regarding parallel execution of all atomic services. In order to execute the workflow with optimal parallelism, we define an configuration file to represent parallel

```
<deployment>
    <service>
        <inputQueue endpoint="inputQueue" size="DOP" />
        <topDescriptor>
            <import location="TwoHopTransDescriptor.xml" />
        </topDescriptor>
        <analysisEngine async="true" key="TwoHopTransDescriptor">
            <delegates>
                <analysisEngine key="FirstTransDescriptor">
                    <scaleout numberOfInstances="DOP" />
                </analysisEngine>
                <analysisEngine key="SecondTransDescriptor">
                    <scaleout numberOfInstances="DOP" />
                </analysisEngine>
            </delegates>
        </analysisEngine>
    </service>
</deployment>
```

Figure 5.8: Example of a parallel execution deployment

execution deployment for the workflow. This is an xml file specifying information to be interpreted by the workflow engine. Figure shows an example of a parallel execution deployment file of the two-hop translation workflow specifying by an pipeline representation file “TwoHopTransDescritor.xml”. This workflow consists of two translation annotators. The workflow will be deployed as a service with an input queue. The size of the input queue determines the number of data partitions can be processed in parallel. Annotators in the workflow are processed asynchronously to support pipelined execution. The number of instances of each annotator is set as DOP of the workflow. With this deployment file, the workflow engine will initiated multiple threads of each annotator to process multiple data partitions concurrently.

Parallel Processes Allocation.

Suppose there are k workflow $w_1, w_2, ..., w_k$ defined as follows: $w_1 = \{s, s_{12}, ..., s_{1l}\}$, $w_1 = \{s, s_{22}, ..., s_{2m}\}$, ... Using our prediction model, we can create formulae to calculate execution time of the workflows:

$$f_1(n) = prediction\_fomula_1(f_s(n), f_{s_{12}}(n), ...f_{s_{1m}}(n))$$
$$f_2(n) = prediction\_fomula_2(f_s(n), f_{s_{22}}(n), ...f_{s_{2m}}(n))$$
$$\cdots$$

These workflows have a binding to the same service s. In the case that these workflows are not invoked in parallel, the optimal DOP of each workflow is estimate by using our proposed prediction model. $p_{opt_1}, p_{opt_2}, ..., p_{opt_k}$ are optimal DOPs of $w_1, w_2, ..., w_3$ respectively. Consider the case that k workflows are invoked concurrently with optimal DOPs. In this case, the number of concurrent requests sent to service s is: $p_{opt_1} + p_{opt_2} + ... + p_{opt_k}$ (if s belongs to a conditional branch of one or more workflows, this equation is calculated differently). If parallel execution policy of the service s is restriction, performance of the service remains stable when number of concurrent requests to the service exceeds a specified number. In this case, executing workflows with optimal DOPs still gains optimal performance improvement for all workflows. However, if the parallel execution policy of the service s is penalty policy, since the number of concurrent requests sent to s exceeds the penalty point, the performance of service s goes down. This may cause execution time of workflows increase. In order to attain optimal reduction gain of execution time of all workflows, the system should allocates suitable numbers of parallel processes for workflows, let's say $(p_1, p_2, ..., p_k)$.

$P = \sum_{1}^{k} p_i$ is number of parallel requests sent to service s. These values are determined so that $\sum_{1}^{k} f_i(p_i)$ is minimized, where:

$$f_1(p_1) = prediction\_fomula_1(f_s(P), f_{s_{12}}(p_1), ...f_{s_{1m}}(p_1))$$
$$f_2(p_2) = prediction\_fomula_2(f_s(P), f_{s_{22}}(p_1), ...f_{s_{2m}}(p_1))$$
$$...$$

After determining the optimal parallel processes allocation, the system dynamically update parallel execution deployment file of each workflow. Finally, workflow execution engine interprets new parallel execution deployment configuration and execute the workflows.

5.5 Experiment

In this experiment we try to evaluate the impact of the parallel control architecture in maintaining the optimal parallel execution of workflows. We conduct experiments on the composite services introduced in the scenario.

- The first two-hop translation service (w_1) is sequential composition of two translation services. A client sends a request and binding information to translate a document of 500 paragraphs from Japanese to Vietnamese via English. The binding information is as follows: the first translator is bound to J-Server translation service, and the second translator is bound to Google translation service.

  J-Server and Google translation services are analysed to determine parallel execution policies. J-Server has slow-down and restriction policies with slow-down point $p_{js} = 4$, restriction point $p_{jr} = 12$. While Google translation service employs slow-down and penalty policies with slow-down point $p_{gs} = 4$, penalty point $p_{gp} = 8$. Apply the policy model execution time of J-Server translation service and Google translation service are calculated with $f_{jserver}(n)$ and $f_{google}(n)$. Suppose that the document is split into $M_1 = 500$ independent partitions, n partitions are sent to the composite service in parallel. Using the prediction model for sequential structure (see section 4.2.3), the execution time of the two-hop translation service is predicted by following equations:

$$f_1(n) \approxeq max(f_{jserver}(n), f_{google}(n)) + \frac{min(f_{jserver}(n), f_{google}(n))}{\lceil M_1/n \rceil}$$

The estimated optimal DOP is $p_{opt1} = 12$. With this DOP, the predicted minimum execution time is 22,230 milliseconds (actual execution time is 22,993 milliseconds).

- The second two-hop translation service contains sequential structure and conditional structure. A client send a request and binding information to translate another Japanese agricultural document containing 500 paragraphs. A half of the document (250 paragraphs), containing information about rice, is translated Vietnamese via English. The other half, containing information about fertilizer, is translate to French (Fr) via English. The binding information is as follows: the first translator (to translate whole document from Ja - En) is bound to Baidu translation service. The second translator (to translate a half of document from En - Vi) is bound to Google translation service.

The third translator (to translate a half of document from En -Fr) is bound to Bing translation service. The concrete translation services are analysed, following are parallel execution policies of the services: Baidu has slow-down and restriction policies with slow-down point $p_{bas} = 6$ and restriction point $p_{bar} = 12$, Bing employs slow-down and restriction policies with slow-down point $p_{bis} = 6$ and restriction point $p_{bir} = 14$. $f_{baidu}(n)$, $f_{bing}(n)$, and $f_{google}(n)$ are formulae to calculate execution time of Baidu, Bing and Google translation services respectively. Suppose that the document is split into $M_2 = 500$ partitions and n concurrent are processed concurrently. Using our prediction model for sequential structure and conditional structure we form a formula to predict execution time of the composite service as following:

$$\begin{aligned} f_2(n) \approxeq \max(f_{baidu}(n), max(f_{google}(n/2) + f_{bing}(n/2))) \\ + \frac{min(f_{baidu}(n), max(f_{google}(n/2) + f_{bing}(n/2))}{\lceil M_2/n \rceil} \end{aligned}$$

Using this equation, the estimated optimal DOP of the composite service is $p_{opt2} = 28$. With this DOP, the minimum execution time of the composite service is 22,230 milliseconds (actual execution time is 23,530 milliseconds). Total time for the two composite services finish processing the two documents is 45,460 milliseconds.

The system generates parallel execution deployment file for each composite service with the estimated optimal DOP. We create a client invoking the two composite services to process the two documents in parallel. Google translation service is invoked in both workflows, in this case Google translation service need to serve $26(12+28/2)$ concurrent requests. The execution time of composite services in this case are: $f_1(12) = 52,584$ milliseconds, and $f_2(28) = 47,645$ milliseconds. The execution time has increased approximately 136% for the first composite service and 102% for the second one compared to the optimal case of invoking single workflow. Total time for two composite service finish processing the two documents is 52,584 milliseconds.

The system re-calculates DOPs of composite services in order to attain optimal reduction gains of execution time. Using equation in previous section, new values of DOPs, p_1 and p_2, are determined so that: $f_1(p_1) + f_2(p_2)$ is minimized. Where $f_1(p_1)$ and $f_2(p_2)$ are adjusted as following:

$$\begin{aligned} f_1(p_1) &= max(f_{jserver}(p_1), f_{google}(p_1 + p_2/2)) \\ &\quad + \frac{min(f_{jserver}(p_1), f_{google}(p_1 + p_2/2))}{\lceil M_1/p_1 \rceil} \\ f_2(p_2) &= \max(f_{baidu}(p_2), max(f_{google}(p_2/2 + p_1) + f_{bing}(p_2/2))) \\ &\quad + \frac{min(f_{baidu}(p_2), max(f_{google}(p_2/2 + p_1) + f_{bing}(p_2/2))}{\lceil M_2/p_2 \rceil} \end{aligned}$$

Solving this problem, the system determines $p_1 = 6$, $p_2 = 20$. The system updates parallel execution deployment of each composite service with new value of DOP. Execution time of composite are: $f_1(6) = 31,083$ milliseconds and $f_2(20) = 24,458$ milliseconds. With the new value of DOPs, the execution time only increases 40% for the first composite service and 4% for the second composite service. Total time for the two services finish processing the two documents is 31,083 milliseconds.

Table 5.1: Summary of experiment results

Composite service	Invoking two composite services sequentially			Invoking two composite services in parallel					
				Without parallel process allocation			Use parallel process allocation		
	Opt. DOP	Execution time (ms)	Total time	Opt. DOP	Execution time (ms)	Total time	Opt. DOP	Execution time (ms)	Total time
w_1	12	22230	45760	12	52584	52584	6	31083	31083
w_2	28	23530		28	47645		20	24458	

Table 5.1 shows summary of performance improvement of the two example of composite services when using our proposed architecture. The experiment results show that, the parallel control architecture significantly increase the efficiency of parallel execution of composite service. Based on parallel execution policies of atomic services, the architecture maintains an optimal degree of parallelism of each composite service.

5.6 Conclusion

In this chapter we presented the implementation of an architecture which uses our propose policy model and prediction model to control parallel execution of composite service in SOA platforms. The architecture can serve as a middle-ware for SOA platforms to support and control parallel execution of composite services. Typically we integrated this architecture as an extended component for the Language Grid platform. We addressed the following two implementation issues in realize this system:

Support parallel and pipelined execution for the workflow engine

We adapted a pipeline execution engine as a workflow engine to support parallel and pipelined execution of composite services. As a case study, we adapted UIMA pipeline engine as a workflow engine in the Language Grid.

Control parallel execution of composite services

We defined a deployment file to represent parallel execution of a composite service. This file specifies degree of parallelism (DOP) of the composite service. The workflow engine interprets the file to execute the composite service with the specified DOP. By specifying suitable DOP, the system can control parallel execution of composite services to attain the optimal reduction gain of execution time.

This architecture helps SOA platforms dynamically control parallel execution of composite services, based on parallel execution policies of all atomic services, to attain optimal performance improvement. The architecture can significantly improve parallel execution efficiency of composite services.

Chapter 6

Conclusion

6.1 Contributions

The thesis presented three contributions toward a policy-aware parallel execution control system for enhancing parallel execution efficiency of composite services. The first is the proposal of policy model which is used to capture parallel execution policies of web services. The second is a prediction model to predict performance of composite services with regard to all atomic services' policies. The last is a policy-aware parallel execution control architecture that serves as a middle-ware for SOA platforms to support and control parallel execution of composite services to attain optimal execution time reduction. Moreover, we have integrated this architecture as an extended component for the Language Grid platform. We, in this section, review these contributions. After that, we will describe few areas for future research.

1. In web service environments, services are provided to users by various service providers. When invoking services with parallel execution, service users may observe different performance improvement behaviors (patterns) of different services. The reasons for the performance improvement patterns may vary such as providers' computing resources, the services' implementation or policies of service providers. However, service users normally have no clear image about the reasons causing the performance improvement patterns. To be simple, from the view of service users, we regarded the performance improvement behavior of a service as the service's policy. We proposed a policy model to capture different service policies. Three policies have been observed and modelled: Slow-down policy, restriction policy and penalty policy. The model is designed to be simple to be used in predicting parallel execution performance of composite services. The evaluation results revealed that our model can accurately capture parallel execution policies of web services.

2. In service composition, many works have been proposed to optimize QoS of composite services, no studies considered parallel execution policies of atomic services to find optimal parallelism for a composite service. We proposed an novel prediction model which embeds parallel execution policies of atomic services into formula to calculate performance of composite services under parallel execution. We also considered different workflow structures of composite services in creating the formulae. From the calculation, we can estimate the optimal parallelism of a composite service, where the service attain the best performance improvement. The experiment results showed

that our prediction model has good accuracy in predicting the optimal parallelism of composite services.

3. To enhance the efficiency of parallelism in SOA platforms, regarding parallel execution policies of services, we need to control parallel execution of services to gain optimal performance improvement. We designed a policy-aware parallel execution control architecture to support and control parallel execution of composite services. By integrating a pipeline engine and a workflow engine, our proposed architecture able to support parallel and pipelined execution to speed-up execution of composite services. As atomic services can be dynamically bound to a composite service, from the binding information our architecture analyzes policies of atomic services and dynamically adjust the optimal degree of parallelism of the composite service based on the prediction. To deal with multiple uses of one atomic service in different workflows, to maintain the optimal gain of execution time reduction, the architecture use the prediction model to allocate suitable parallel processes for each workflows. As the implementation, we integrated this architecture as an middle-ware for the Language Grid platform[1]. Experiment results show that this parallel execution control architecture significantly improve efficiency parallelism of composite services in the Language Grid.

[1]The Language Grid: http://langrid.org/

6.2 Future Direction

With the new idea on using parallel execution policies in web service composition as presented in this thesis, following future research directions are suggested:

- Extending coverage of the policy model

  In this thesis, we defined three types of parallel execution policies of web services which are slow-down policy, restriction policy and penalty policy. However these three types of policies may not correctly cover all types of web service policies. There is possibility of having a scale-out policy if we use paid services, if user pay more they will get higher performance improvement when using parallel execution. Or there is also a policy specify limitation of number of concurrent requests on a certain amount of time. To make our model more rigorous, more analyses on larger number of web services and more parameters for parallel execution such as number of concurrent requests per second are needed.

- Parallel execution policies on multiple QoS criteria

  In this thesis, we mainly focus on analysing execution time of services in modeling the parallel execution policies. However, there are also changing behaviors of different QoS criteria such as cost, and reputation when we invoke a service with parallel execution. One typical example is that, if a user invoke the service with high number of concurrent requests then user need to pay more. There is also trade-off between execution time and cost when using parallel execution. Consider multiple QoS criteria in modeling parallel execution policies makes the model more accurate and useful for composing and optimizing composite services.

- Dynamic parallel execution policies

  In designing the proposed model, we assumed that parallel execution policy of one service is static. This means that, the limit on the number of concurrent process for the service does not change even the input data size is changed. However, in cloud environments, it seems highly likely that service providers will dynamically change their policies in response to requests with different sizes. To capture the policies more correctly, the model need to consider the dynamic changes of the service policies.

Publications

Journal

1. **Mai Xuan Trang**, Yohei Murakami, and Toru Ishida. Policy-Aware Service Composition: Predicting Parallel Execution Performance of Composite Services. *IEEE Transactions on Services Computing (TSC), special issue on Cloud Services Meet Big Data*, 2015.

International Conferences

1. **Mai Xuan Trang**, Yohei Murakami, Donghui Lin, and Toru Ishida. Interoperability between Service Composition and Processing Pipeline: Case Study on the Language Grid and UIMA. In *Proceedings of the* 6*th* *International Joint Conference on Natural Language Processing (IJCNLP), pp. 1052-1056*, Nagoya, 2013. (short paper).

2. **Mai Xuan Trang**, Yohei Murakami, Donghui Lin, and Toru Ishida. Integration of Workflow and Pipeline for Language Service Composition. In *Proceedings of the* 9*th* *Language Resources and Evaluation Conference (LREC 2014), pp. 3829-3836*, Iceland, 2014.

3. **Mai Xuan Trang**, Yohei Murakami, and Toru Ishida. Policy-Aware Optimization of Parallel Execution of Composite Service. In *Proceedings of the* 12*th* *IEEE International Conference on Services Computing (SCC), pp. 106-113*, New York, 2015. (Best Paper Award)

4. **Mai Xuan Trang**, Yohei Murakami, and Toru Ishida. Modeling Parallel Execution Policies of Web Services. In *Proceedings of the* 6*th* *EAI International Conference on Cloud Computing*, Korea, 2015.

Workshops

1. **Mai Xuan Trang**, Yohei Murakami, Taketo Sasaki and Toru Ishida. Language Mashup: Personalized Platform for Language Service Composition. In *Proceedings of the IEICE Technical Report. Artificial Intelligence and Knowledge Processing*, IEICE Tech. Rep., vol. 113, no. 441, AI2013-44, pp. 41-46, Osaka, 2014.

2. **Mai Xuan Trang**, Yohei Murakami, and Toru Ishida. A Policy-Aware Parallel Execution Control Mechanism for Language Application. *The Second International Workshop on Worldwide Language Service Infrastructure*, Kyoto, 2015.

Other Publications

1. Visit Hirankitti and **Mai Xuan Trang**. A Meta-reasoning Approach for Reasoning with SWRL Ontologies. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, pp. 112-117, Hong Kong, 2011.

2. Visit Hirankitti and **Mai Xuan Trang**. A Meta-logical Approach for Reasoning with an OWL 2 Ontology. *Journal of Ambient Intelligence and Humanized Computing*, 3(4), pp. 293-303, 2012.

3. Visit Hirankitti and **Mai Xuan Trang**. Reasoning on OWL2 Ontologies with Rules Using Metalogic. In *Electrical Engineering and Intelligent Systems*, pp. 95-108, Springer New York, 2013.

Bibliography

[Al-Moayed and Hollunder, 2010] Al-Moayed, A. and Hollunder, B. (2010). Quality of service attributes in web services. In *Proceedings of the Fifth International Conference on Software Engineering Advances (ICSEA 2010)*, pages 367–372. IEEE.

[Allen et al., 2003] Allen, G., Goodale, T., Radke, T., Russell, M., Seidel, E., Davis, K., Dolkas, K. N., Doulamis, N. D., Kielmann, T., Merzky, A., et al. (2003). Enabling applications on the grid: A gridlab overview. *International Journal of High Performance Computing Applications*, 17(4):449–466.

[Altintas et al., 2005] Altintas, I., Birnbaum, A., Baldridge, K. K., Sudholt, W., Miller, M., Amoreira, C., Potier, Y., and Ludaescher, B. (2005). A framework for the design and reuse of grid workflows. In *Scientific Applications of Grid Computing*, pages 120–133. Springer.

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference*, pages 483–485. ACM.

[Andrews et al., 2003] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., et al. (2003). Business process execution language for web services.

[Ardagna and Pernici, 2007] Ardagna, D. and Pernici, B. (2007). Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369–384.

[Bel, 2010] Bel, N. (2010). Platform for automatic, normalized annotation and cost-effective acquisition of language resources for human language technologies. panacea. *Procesamiento del Lenguaje Natural*, 45:327–328.

[Berthold et al., 2008] Berthold, M. R., Cebron, N., Dill, F., Gabriel, T. R., Kötter, T., Meinl, T., Ohl, P., Sieb, C., Thiel, K., and Wiswedel, B. (2008). Knime: The konstanz information miner. In *Data analysis, machine learning and applications*, pages 319–326. Springer.

[Bramantoro and Ishida, 2009] Bramantoro, A. and Ishida, T. (2009). User-centered qos in combining web services for interactive domain. In *Proceedings of the Fifth International Conference on Semantics, Knowledge and Grid (SKG 2009)*, pages 41–48. IEEE.

[Bray et al., 1998] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (1998). Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16.

[Canfora et al., 2008] Canfora, G., Di Penta, M., Esposito, R., and Villani, M. L. (2008). A framework for qos-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769.

[Cardoso et al., 2004] Cardoso, J., Sheth, A., Miller, J., Arnold, J., and Kochut, K. (2004). Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308.

[Chinnici et al., 2007] Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. (2007). Web services description language (wsdl) version 2.0 part 1: Core language. *W3C recommendation*, 26:19.

[Deelman et al., 2005] Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., et al. (2005). Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237.

[Ferrucci et al., 2010] Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J., et al. (2010). Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79.

[Ferrucci and Lally, 2004] Ferrucci, D. and Lally, A. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

[Goecks et al., 2010] Goecks, J., Nekrutenko, A., Taylor, J., et al. (2010). Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86.

[Gordon et al., 2006] Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 151–162. ACM.

[Goto et al., 2011] Goto, S., Murakami, Y., and Ishida, T. (2011). Reputation-based selection of language services. In *Proceedings of the 2011 IEEE International Conference on Services Computing (SCC)*, pages 330–337. IEEE.

[Guan et al., 2006] Guan, Y., Ghose, A. K., and Lu, Z. (2006). Using constraint hierarchies to support qos-guided service composition. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS'06)*, pages 743–752. IEEE.

[Gudgin et al., 2003] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., and Lafon, Y. (2003). Simple object access protocol (soap) 1.2. *World Wide Web Consortium.*

[Hassine et al., 2006] Hassine, A. B., Matsubara, S., and Ishida, T. (2006). A constraint-based approach to horizontal web service composition. In *The Semantic Web-ISWC 2006*, pages 130–143. Springer.

[Hassine et al., 2011] Hassine, A. B., Matsubara, S., and Ishida, T. (2011). Horizontal service composition for language services. In *The Language Grid*, pages 53–67. Springer.

[Humbetov, 2012] Humbetov, S. (2012). Data-intensive computing with map-reduce and hadoop. In *Proceedings of the 6th International Con-*

ference on Application of Information and Communication Technologies (AICT 2012), pages 1–5. IEEE.

[Ide and Romary, 2009] Ide, N. and Romary, L. (2009). Standards for language resources. *arXiv preprint arXiv:0911.1842*.

[Ishida, 2011] Ishida, T. (2011). *The language grid: Service-oriented collective intelligence for language resource interoperability*. Springer Science & Business Media.

[Jaeger et al., 2004] Jaeger, M. C., Rojec-Goldmann, G., and Mühl, G. (2004). Qos aggregation for web service composition using workflow patterns. In *Proceedings of the 8th International Enterprise distributed object computing conference (EDOC 2004)*, pages 149–159. IEEE.

[Leymann et al., 2001] Leymann, F. et al. (2001). Web services flow language (wsfl 1.0).

[Lin et al., 2012] Lin, D., Shi, C., and Ishida, T. (2012). Dynamic service selection based on context-aware qos. In *Proceedings of the Ninth International Conference on Services Computing (SCC 2012)*, pages 641–648. IEEE.

[Martin et al., 1997] Martin, R. P., Vahdat, A. M., Culler, D. E., and Anderson, T. E. (1997). *Effects of communication latency, overhead, and bandwidth in a cluster architecture*, volume 25. ACM.

[Missier et al., 2010] Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., and Goble, C. (2010). Taverna, reloaded. In *Scientific and Statistical Database Management*, pages 471–481. Springer.

[Murakami et al., 2011] Murakami, Y., Lin, D., Tanaka, M., Nakaguchi, T., and Ishida, T. (2011). Service grid architecture. In *The Language Grid*, pages 19–34. Springer.

[Oliveira et al., 2012] Oliveira, D., Ogasawara, E., Ocaña, K., Baião, F., and Mattoso, M. (2012). An adaptive parallel execution strategy for cloud-based scientific workflows. *Concurrency and Computation: Practice and Experience*, 24(13):1531–1550.

[Papazoglou, 2003] Papazoglou, M. P. (2003). Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, pages 3–12. IEEE.

[Pautasso and Alonso, 2006] Pautasso, C. and Alonso, G. (2006). Parallel computing patterns for grid workflows. In *Workshop on Workflows in Support of Large-Scale Science*, pages 1–10. IEEE.

[Preist, 2004] Preist, C. (2004). A conceptual architecture for semantic web services. In *The Semantic Web–ISWC 2004*, pages 395–409. Springer.

[Raicu et al., 2012] Raicu, I., Foster, I., Zhao, Y., Szalay, A., Little, P., Moretti, C. M., Chaudhary, A., and Thain, D. (2012). Towards data intensive many-task computing.

[Sun and Chen, 2010] Sun, X.-H. and Chen, Y. (2010). Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188.

[Tallent and Mellor-Crummey, 2009] Tallent, N. R. and Mellor-Crummey, J. M. (2009). Effective performance measurement and analysis of multithreaded applications. In *ACM Sigplan Notices*, volume 44, pages 229–240. ACM.

[Taylor et al., 2014] Taylor, I. J., Deelman, E., Gannon, D. B., and Shields, M. (2014). *Workflows for e-Science: scientific workflows for grids*. Springer Publishing Company, Incorporated.

[van Der Aalst et al., 2003] van Der Aalst, W. M., Ter Hofstede, A. H., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distributed and parallel databases*, 14(1):5–51.

[Vanhoutte, 2004] Vanhoutte, E. (2004). An introduction to the tei and the tei consortium. *Literary and linguistic computing*, 19(1):9–16.

[Yu and Bouguettaya, 2008] Yu, Q. and Bouguettaya, A. (2008). Framework for web service query algebra and optimization. *ACM Transactions on the Web (TWEB)*, 2(1):6.

[Yu et al., 2007] Yu, T., Zhang, Y., and Lin, K.-J. (2007). Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6.

[Zeng et al., 2003] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., and Sheng, Q. Z. (2003). Quality driven web services composition. In *Proceedings of the 12th international conference on World Wide Web*, pages 411–421. ACM.

[Zeng et al., 2004] Zeng, L., Benatallah, B., Ngu, A. H., Dumas, M., Kalagnanam, J., and Chang, H. (2004). Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327.