PRINCIPLES FOR THE DESIGN AND OPERATION OF ELASTIC SCIENTIFIC APPLICATIONS ON DISTRIBUTED SYSTEMS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by
Dinesh Rajan Pandiarajan

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2015

PRINCIPLES FOR THE DESIGN AND OPERATION OF ELASTIC SCIENTIFIC APPLICATIONS ON DISTRIBUTED SYSTEMS

Abstract

by

Dinesh Rajan Pandiarajan

Scientific applications often harness the concurrency in their workloads to partition and operate them as independent tasks and achieve reasonable performance. To improve performance at scale, the partitions are operated in parallel on large pools of resources in distributed computing systems, such as clouds, clusters, and grids. However, the exclusive and on-demand deployment of applications on these platforms presents challenges. The target hardware is unknown until runtime and variable between deployments when applications are deployed on these platforms. So operating parameters such as the number of partitions and the instances to provision for execution must be determined at runtime for efficient operation.

In this work, I build and demonstrate *elastic applications* to provide the desired characteristics for operation on distributed computing systems. I present case-studies of elastic applications from different scientific domains and draw broad observations on their design and the challenges to their efficient operation. I develop and evaluate techniques at the middleware and the application layer to achieve efficient operation. In effect, the presented techniques create *self-operating elastic applications* that dynamically determine the partitions of their workloads and the scale of resources to utilize. I conclude by showing that self-operating applications achieve high time- and cost-efficiency in their deployed environments in distributed computing systems.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I thank my advisor, Prof. Douglas Thain, for his technical guidance and inputs on the work performed in this dissertation. He taught me valuable lessons on performing research, building software for an active user base, and communicating ideas. In addition, I got to observe and learn the nuances of effective collaboration, public speaking, and leadership by standing close to him.

I thank Prof. Scott Emrich and Prof. Jesus Izaguirre for their guidance and insights during collaborations on building and operating large scientific applications. I also thank them for taking the time to serve on my dissertation committee.

I thank Prof. Aaron Striegel for serving on my dissertation committee and for showing me to my first job.

I thank Haiyan Meng, Nicholas Hazekamp, Ben Tovar, Hoang Bui, Li Yu, Peter Bui, Shengyan Hong, and Steve Kurtz for the stimulating and vibrant work environment in Cushing 222. I also thank Patrick Donnelly for being an honest and good friend in the many afternoons spent dissecting ideas and thoughts on an assortment of topics.

I am forever thankful to my parents, Pandia and Shoba Rajan, who instilled in me a strong belief that learning is a lifelong process and that it must be pursued at all times. I would not have been able to get to where I am without their sacrifices and support.

I am grateful to my sister, Prathiba Meenakshi, for being a non-judgemental and patient sounding board.

I thank my in-laws, Tim and Andrea O'Connor, for their moral support and many nourishments. They made several things easier during graduate school.

I am also grateful to my grandparents who taught me a number of lessons that I found useful in graduate school - Mohan and Mrinalini Rajan for teaching the importance of approaching problems with a positive and determined mindset, and Umashankar and Shanti Shankar from whom I learned the benefits of a healthy and refreshed mind.

Finally, I am thankful to my lovely wife, Sara Rajan, for her endless love, patience, and support. She played a huge role in both my returning to graduate school and completing it. Her encouragement and support enabled me to stay focused throughout graduate school.

CHAPTER 1

INTRODUCTION

1.1 Concurrent Scientific Applications on Distributed Systems

The scale and complexity of software in scientific- and data-oriented fields continues to increase to sustain the advancement of knowledge in their domains. Examples can be observed in a variety of scientific fields such as bioinformatics [15, 104], molecular dynamics [85, 107], data mining [130], fluid dynamics [99], genetic algorithms [31], and biometrics [82]. In order to achieve reasonable performance at large scales, these applications typically incorporate or exploit *concurrency* in their workloads. That is, these applications partition the workloads into concurrent pieces and operate them independently on allocated resources.

Common approaches used for enabling the concurrent operation of scientific applications include design patterns [35, 36, 39], programming abstractions [43, 62, 100], language support [27, 121], runtime support [24], and middleware [108]. A well-known example is the Message Passing Interface (MPI) [43] paradigm that partitions the workload as concurrent processes and operates them simultaneously on a equivalent number of CPUs. A more recent example is Hadoop that partitions the workload into concurrent tasks [35, 39] and operates them on multiple compute nodes.

Even with the exploitation of concurrency during execution, these applications have grown to surpass the compute and storage capacity of a single or a small network of nodes. Their resource needs are often satisfied only by the aggregate capacity of hundreds to thousands of compute nodes [64]. In other words, the infrastructure for the operation of concurrent scientific applications are only available in large distributed computing systems [7, 16, 111]. These systems provide the hardware, resources, and interfaces for deploying and operating scientific applications at scale. The current generation of the distributed computing systems can be classified as clusters, clouds, and grids.

Clusters. Clusters are a large pool of interconnected resources that are tightly controlled for access and usage by the organizations maintaining them. Access to the resources is often provided through batch scheduling systems such as SGE [49], PBS [55], Maui [63], etc. Clusters are usually found in university campuses and research laboratories. Also, data centers maintained by commercial organizations for internal use are strikingly similar to clusters in how they are operated and accessed. It is common to find the hardware, capacity, capabilities, and performance of resources in a cluster to be homogeneous and fixed. As a result, operation on clusters is limited to the operating environment defined and configured on the resources. Clusters provide dedicated access to resources for the requested duration of operation. However, the duration of operation using resources in clusters is typically bounded by a globally enforced interval (often 48 hours).

Clouds. Operation using resources in grids and clusters are typically restricted to the participants, members, and stakeholders of the organizations maintaining them. The rapid emergence of cloud computing over the last decade has addressed this limitation and democratized access to large scale computing. Cloud computing providers, such as Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS), provide public and on-demand access to resources at scale using a metered pricing model [47, 124]. These providers also offer access to resources of various types and sizes by grouping them into tiers according to the size, capacity, and capabilities of the resources. Further, the operating environment in cloud platforms can be completely configured to suit the needs of the applications. It is important to note the deployment of applications on cloud platforms incurs monetary costs to the operators who provision and maintain resources for the operation of the applications.

Grids. Grids are a federation of resources aggregated across multiple clusters, campuses, data centers, and organizations. They are built and operated to enable inexpensive access to large scales of resources to the contributing organizations who otherwise lack access to such scales in the resources directly available under their ownership. Grids are focused on providing a seamless interface to geographically distributed and heterogeneous resources with different access control policies, services, and interfaces for operation. Examples of grids include Condor [111], Open Science Grid [91], and BOINC [20], and Folding@Home [102]. Grids typically operate by harnessing the idle and unused resources in the federation to run large computations. They allocate idle cycles or capacity as resources available for operation and terminate them when an higher-priority operator, such as the owner of those resources, begins consumption. As a result, applications operating on grids must be prepared for the addition and termination of resources at a moment's notice.

The infrastructure offered in these platforms consist of compute, storage, and network resources that are often virtualized to enable simultaneous sharing of resources. These platforms are also considered to provide services for mapping jobs onto provisioned resources and balancing load across the available resources. Users can harness these platforms through the interfaces they offer for requesting, configuring, and terminating the allocation of resources, and for submitting processes for execution on the allocated resources.

As the scale and complexity of scientific applications continues to increase, the number of providers of clusters, clouds, and grids, and the pool of resources offered through them have continued to increase to meet the demand. This trend is beneficial for end-users of the applications since it grants them multiple options for deploying the applications at the scales they require or desire. Further, these platforms enable end-users to provision resources on-demand and maintain them only for the duration of use. In addition, the direct access to resources allows end-users to provision resources across multiple platforms or providers to build the desired operating environment for the applications. For instance, an end-user might provision resources from a cluster and complement it with resources provisioned from multiple cloud providers to accelerate the execution when desired.

In summary, the current generation of distributed computing systems have introduced the ability to achieve exclusive deployments at large scales by provisioning and maintaining resources exclusively for a single invocation of the application.

1.2 Problem Statement

The advancements provided by the current generation of distributed computing systems described above have also introduced a new set of challenges. The access to a variety of resource providers has resulted in a wide variety of resource and hardware configurations on which the application can be deployed and operated. This is further compounded when end-users federate resources across multiple platforms to achieve the scales needed for operation. That is, the target hardware and operating environment of such applications are unknown until runtime and prone to vary in each invocation of the same application. This presents challenges since existing approaches to the design and operation of large concurrent applications assume prior knowledge of the target environment during their design [24, 43, 108].

The challenges to the design and operation of concurrent scientific applications on the current generation of distributed computing systems can be stated as follows:

- How should applications be designed and partitioned on platforms when their target hardware is unknown and variable at the time of their design?
- How should applications determine and express their resource requirements in terms of the scale of compute instances required for efficient operation?

In addition, platforms such as clouds charge the end-users directly for the resources they provision and use. Therefore, the cost of operation must be considered so the end-users can operate the applications in a cost-efficient manner. The challenges described above can be extended and summarized as follows:

How should applications be built and operated so they achieve high cost-efficiency during operation at scale when their target hardware is unknown and variable at the time of their design?

1.3 Overview of Existing Approaches

I begin the search for a solution that addresses these challenges with a discussion of the deployment and operation model of concurrent applications on distributed computing systems as illustrated in Figure 1.1. In this figure, the *developers* are the entities that design and implement the algorithms that define the structure of the workload of the applications. The developers use and incorporate various techniques in the applications to partition their workloads so they can be operated concurrently. On the other hand, the *operators* are the entities that operate the applications by provisioning resources and deploying the application on these resources. The operators provide the inputs to the application and utilize the results returned by the application. Note that the operators of scientific applications on distributed computing systems are often the end-users who directly benefit from the operation of the application on the provided inputs.

I proceed to use the model presented in Figure 1.1 to discuss the current state-of-the-art in the development and operation of concurrent scientific applications. This discussion will highlight the assumptions and drawbacks of existing techniques and establish the need for a new class of techniques to address the identified challenges.

The techniques currently employed in the development and operation of large concurrent applications can be presented under three categories:

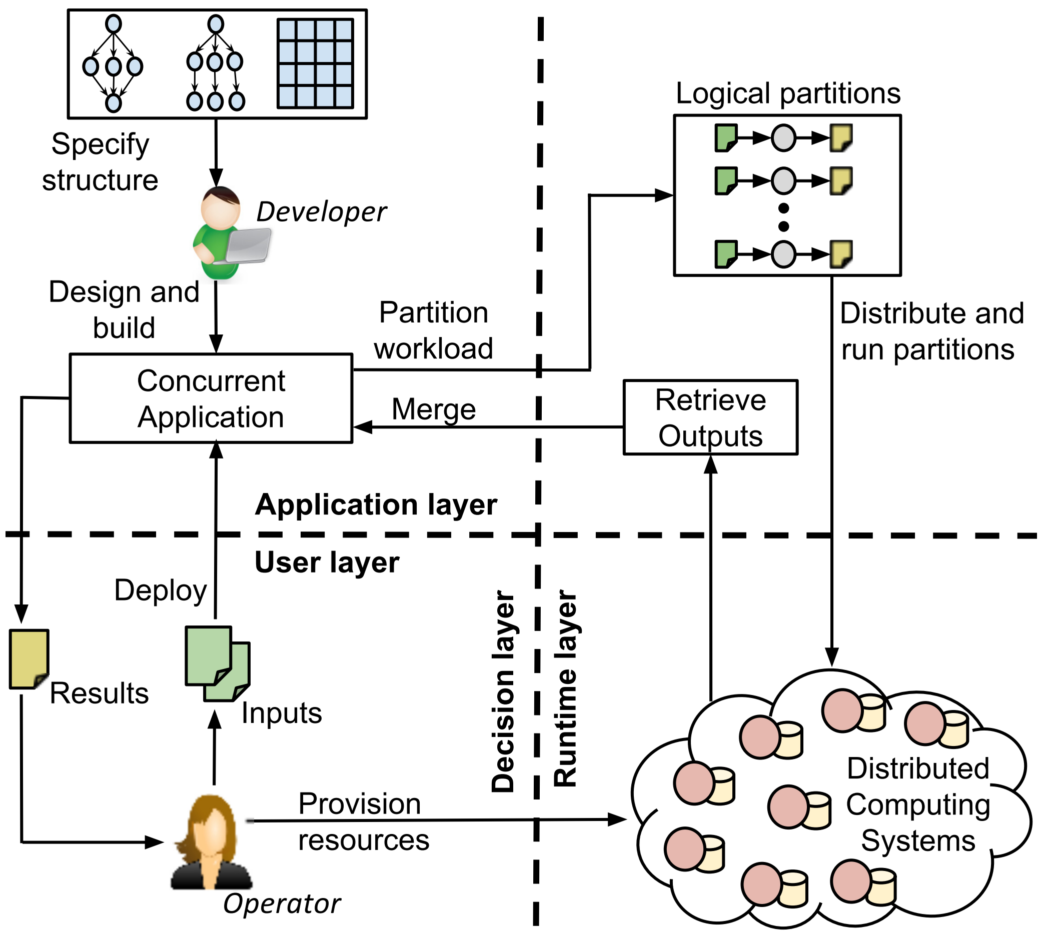


Figure 1.1. The deployment model of concurrent applications on distributed computing systems.

- The developers of the application define the behavior during operation by specifying the target hardware for operation and the runtime behavior on that hardware as part of the design. This allows developers to build and tune the application to specific hardware and achieve efficient operation on them. This approach is beneficial when the application is deployed exactly on the hardware and operating environment assumed during design. Examples of this approach are found in MPI [43] and Cilk [24].

- The operators of the applications specify the resources for operation and tune the operating parameters of the application based on the size, type, and characteristics of the resources being provisioned. This approach is prevalent in environments where there are multiple applications and invocations run on the same pool of provisioned infrastructure. The operators here determine the parameters for the operation of the applications such that the provisioned infrastructure is utilized in a cost-efficient manner. Hadoop [53] employs this approach in requiring the operators of the Hadoop cluster to define and specify the performance and reliability characteristics for the operation of applications on the cluster.

- The middleware or runtime system between the resources and the applications takes control of the operation of the application and operates the application in a manner it defines as efficient. SEDA [119] is a popular example of this approach. In these approaches, the middleware assumes certain characteristics of the workload or requires these characteristics to be communicated at runtime. The knowledge of the workload characteristics is required for the middleware to make intelligent decisions for their efficient operation.

In summary, these approaches do not support the efficient operation of applications on hardware unknown during design, or exclusive deployments for each invocation, or execution of arbitrary workloads.

1.4 Case for Applications as Operators

To address the challenges described in Section 1.2, I propose and demonstrate that scientific applications must (1) be built as elastic applications and (2) act as operators in their deployed environments in distributed computing systems.

In this work, I show that applications built and run as *elastic applications* are fault-tolerant, adaptive to the resources available for operation, portable across platforms, and simultaneously operate on heterogeneous hardware and operating environments. These characteristics are essential and beneficial for successful operation on current distributed computing environments characterized by metered, on-demand, and exclusive deployments on commodity hardware.

In addition, I argue that elastic applications must assume the responsibilities of their operators in *determining the resources to provision for operation, directing the concurrent execution of their workloads, and adapting their execution to the operating conditions*. By serving as their operators, applications can measure the operating conditions and precisely harness knowledge of the current and future needs and characteristics of their workloads to determine the operating parameters for efficient operation. Even when the target hardware is unknown and variable, self-operating applications achieve time- and cost-efficient operation by determining the resource requirements and tuning the concurrent operation of their workloads according to the measured operating conditions.

The key ideas advocated and demonstrated by this dissertation can be summarized as follows:

Self-operating elastic applications achieve time- and cost-efficient operation on distributed computing systems. These applications directly partition the concurrency in their workloads into independent tasks, submit the tasks for simultaneous execution, determine the resources to allocate, and adapt their operation according to the characteristics of the deployed environment.

1.5 Overview of Dissertation

This dissertation studies concurrent scientific applications and the challenges associated with their operation on the current generation of distributed systems comprised of clusters, clouds, and grids. It then addresses the challenges by presenting and evaluating techniques for building self-operating elastic applications that overcome the challenges in achieving time- and cost-efficient operation. It organizes this presentation in the following chapters.

Chapter 2: Related Work. This chapter reviews prior work on the programming paradigms and techniques used for the concurrent operation of large workloads on distributed systems. It also discusses current techniques in the middleware and application layer as well as the principles currently advocated in altering the resource configurations according to the requirements of the applications. This chapter shows why these principles and techniques cannot be extended to scientific applications with fixed-size workloads directly operated by end-users on distributed computing systems.

Chapter 3: Elastic Applications. Large scientific workloads are increasingly deployed at scale on resources with diverse and commodity hardware configurations. The reasons behind this trend extend from the readily available access to the wide variety of resources offered in cloud platforms to the federation of resources from one or more cluster, cloud, or grid computing platforms. This chapter argues the need for *elastic applications* and identifies the desired characteristics for operation on diverse hardware configurations. It experimentally demonstrates the characteristics of an elastic application and their benefits for operation on diverse and dynamically changing operating environments.

Chapter 4: Case Studies of Elastic Applications. This chapter utilizes the established characteristics of elastic applications in the previous chapter to convert a variety of scientific workloads to elastic applications. This chapter shows the conversion of six scientific workloads and describes their elastic implementations in terms of their construction, operation, and the solutions devised to address the challenges encountered during construction and operation. The chapter also derives high-level principles for the design of elastic applications from the experiences in building and operating these applications.

Chapter 5: Middleware Techniques. This chapter formulates the principles for the design of middleware used for the construction and operation of elastic applications. It argues for loosening the guidelines imposed by conventional wisdom on middleware to hide all information about the underlying operating environment. Specifically, the chapter argues that middleware should expose selective information about the underlying operating environment to the applications. The chapter also shows that gains in efficient operation can be achieved when middleware consider and implement strategies for the migration and management of data during operation of the applications on distributed computing systems at scale.

Chapter 6: Application-level Techniques. This chapter presents techniques for self-operating applications that overcome the challenges with achieving cost-efficient operation without knowledge of the operating environments at the time of their design. The chapter shows applications that incorporate the techniques of application-level model, control, and adaptation to achieve cost-efficiency in any deployed environment without intervention from the end-users who provision resources for the application.

Chapter 7: Conclusions. This chapter summarizes the principles and techniques presented in this dissertation for the efficient operation of scientific applications at scale. It shows how the presented techniques for self-operating applications fit among the use of abstractions and agents for the design and operation of applications as advocated by previous work. It concludes the dissertation with a vision for how the presented techniques can be extended to construct fully autonomous distributed computing applications and identifies future work needed to achieve this vision.

CHAPTER 2

RELATED WORK

2.1 Scientific Computing

Scientific applications run resource-intensive computations or analysis that are based on a workload determined by the algorithms they implement and the inputs and configuration parameters supplied at runtime. In the previous decades, the scales required for the operation of these applications were only available in super- and high-performance computing environments [68, 106]. These environments provided access to expensive and high-end hardware with low failure rates and uniform performance profiles. These characteristics of the operating environment enabled the applications to be built and tuned for a specific type of hardware on which they were required or expected to be operated.

The above high-performance computing environments were also characterized by restricted access to the stakeholders/members of the organization maintaining them due to the proprietary setups and expensive infrastructure and maintenance costs. This meant the operation of large scientific applications was available to only those with access to these special and restricted environments. As a result, the developers and operators of these applications were limited and often found it easy to collaborate in building and operating the applications at scale.

The last two decades have seen the rise and growth of distributed computing systems in the form of clusters [49, 55, 63], clouds [3, 16], and grids [7, 91, 111]. These distributed computing systems have democratized access to large arrays of computational resources by offering them for public consumption at low costs. By providing easy and quick access to large-scale resources, these platforms have enabled scientific software to expand in their scale and complexity and enable scientific breakthroughs.

To help navigate the challenges of building and operating applications on different distributed computing systems and their myriad providers, several frameworks and abstractions were invented. Wolski et al, were the first to present such a framework for operating applications simultaneously across multiple distributed computing platforms [122]. More recently, [79] demonstrated a framework that transparently runs applications across heterogeneous resources federated from multiple cloud and grid platforms. There have also been work on building programming paradigms operating scientific and data-intensive applications on distributed computing systems [31, 40, 52, 84].

Other efforts have focused on providing broad lessons learned from their experiences in operating scientific applications on different platforms [59, 73, 125]. The authors in [59] report and evaluate their experiences in running scientific applications in a virtualized cloud environment. They show that the scheduling and communication overheads need to be carefully considered in evaluating the benefits of running scientific applications on a cloud platform. Lu et al, in [73], describe their experiences in running a bioinformatics application on Windows Azure and present best practices for handling the overheads of large scale concurrent operation on cloud platforms.

However, the challenges of efficient operation on distributed computing systems have been largely left to the application designers. This has resulted in piecemeal approaches that target a specific class of applications [88, 128] or an operating environment [10, 32]. In addition, the use of cloud platforms for operation incurs monetary costs to the operators of the application who provision and maintain the necessary resources. This demands applications to consider and achieve cost-efficiency during operation. Further, the detachment between the developers of the applications and the users or operators of the applications presents new challenges that need to be considered and addressed. For example, the operators of the applications might deploy the applications on hardware at a scale that were not known or considered by the developers of the application during design.

2.2 Service-oriented workloads

Several efforts have focused on solving challenges with the optimal operation and provisioning of resources for service-oriented and multi-tenant environments such as web applications [114], e-commerce systems [117], Software-as-a-Service (SaaS) [14], and databases [96]. Service-oriented workloads handle and satisfy requests from end users for access to data that is stored or processed remotely such as web content, databases, web applications, etc. They are characterized by dynamic workloads that are dictated by the presence and demands of end users.

The techniques for service-oriented environments include load prediction and load mapping [123], estimation [101, 131], analysis of previous deployments and loads [117], and monitoring and adapting the provisioned resources according to the needs of the services [90, 114]. The techniques advocate the adaptation of the operating environment to the demands of the services.

Service-oriented workloads are unknown, unpredictable, and determined by external factors such as the current demands of users during runtime. These parameters are often externally monitored to determine and adapt the size of resources allocated for their operation. Further, the economy of operation of these workloads are determined by the operating costs incurred in serving multiple users and guaranteeing the negotiated service level agreements. The characteristics of service-oriented processes are summarized and compared against those of scientific applications characterized by fixed workloads in Table 2.1.

The differences with service-oriented and multi-tenant environments require tech-

TABLE 2.1

KEY DIFFERENCES BETWEEN COMPUTATION-ORIENTED AND SERVICE-ORIENTED PROCESSES

	Computation-oriented	**Service-oriented**
Instances	Multiple instances; one for each user	Single instance; shared across users
Workload	Pre-defined	Unknown and unpredictable
Deployment	Different platforms and environments	Anchored to one platform and environment
Invocation	Multiple times, serves users who invoked	Once, serves multiple disparate users
Need for Scale	Concurrency/parallelism	Load balancing, redundancy, replication
Performance Metric	Operating time and costs	Service level agreements and profit
Operation Lifetime	Finite duration	Prolonged/indefinite
Changes in Operating Environment	Impact throughout lifetime	Impact isolated to specific duration

niques for the cost-efficient deployment and operation of scientific applications with fixed workloads to be considered and devised differently.

2.3 Workload Decomposition

The decomposition and partitioning of workloads into concurrent pieces is a well-studied topic in computer science since its early days [86, 95]. Over the years, this study has extended to scientific workloads and workflows to present techniques for improving their scale and time to completion [30, 67, 109].

The partitioning and decomposition of workloads have been studied in shared execution environments, such as grids and clusters [34, 54]. The work in [34] studies the use of genetic algorithms to determine a scheduling strategy that can be applied to determine the optimal partitions and the resources to provision. Hedayat et al. [54]

describe a distributed workflow management framework for partitioning and operating workflows on multiple clusters. These efforts however do not consider efficient operation as a metric in building and evaluating their solutions.

Recently, a number of efforts have explored novel and unconventional mechanisms to solving the partitioning of workloads in an optimal manner. Agarwal et al. [10] considered a system with a large proportion of recurring jobs and utilized information from prior executions to determine the optimal degree of parallelism to enable during operation. The work in [67] describes the importance of identifying the optimal number of data partitions for MapReduce applications. It presents preliminary insights from an approach combining code and data analysis with optimization techniques.

There have also been efforts that consider the partitioning of a specific class of applications. For instance, the work in [31] studies and establishes a theoretical model to predict execution times and compute the optimal number of workers for genetic algorithms. Another instance is the work in [109] that formulates a graph partitioning algorithm for DAG workflows such that the data transfer overheads between the nodes of the DAG is lowered.

However, the partitioning of workloads to achieve time- and cost-efficient operation in every invocation based on the characteristics of the deployed environment have not been considered and addressed. This work considers the partitioning of workloads in such environments where the applications can be deployed in environments that are unknown and unpredictable during design and vary in each invocation.

2.4 Resource Allocation

The provisioning and allocation of resources for large-scale and resource-intensive applications have been extensively studied in the context of distributed computing [37, 66, 98]. Recently, several efforts have considered the cost-efficient deployment of scientific and data-intensive workloads in cloud platforms [48, 56, 116]. The authors in [48] and [33] present heuristics for the optimal allocation of resources for an arbitrary batch of independent jobs. They also show that the optimal allocation of resources for an arbitrary batch of independent jobs to be NP-Complete and then present heuristics-based algorithms for allocation.

The effort presented in [56] describes scheduling techniques in the middleware for multi-user environments running on cloud resources. Aneka presented in [116] is a middleware that maximizes resource utilization and cost-efficiency by multiplexing the execution of several applications on available resources and dynamically scaling resources when demand exceeds supply. These frameworks require or propose some coordination between the resource providers and the developers and operators of the applications in order to maximize the utilization of the provisioned resources.

Previous efforts have built and applied elaborate models for optimizing the time and cost of operation of applications in certain operating environments in [23, 120]. They utilize feedback from allocated nodes and apply the model to drive resources allocation based on time and cost constraints of the user. The framework in [23] is targeted at deployments in a hybrid environment comprised of resources drawn from a local cluster and a commercial cloud. The framework schedules jobs on the local cluster and provisions resources in the cloud when the capacity at the cluster cannot satisfy the time constraints of the applications. The Conductor framework in [120] presents an abstraction for efficiently deploying MapReduce applications. It includes a model describing the costs and the computation and storage capacities of various instances offered in a cloud platform. The models are applied to select the services that achieve the lowest cost in running the MapReduce application. The applicability of the framework is restricted to the cloud services and instances that are modeled.

Other efforts have studied and built scheduling techniques for finding the optimal number of resources in grid and cloud computing environments for a given workflow under cost constraints [29, 66, 88]. These efforts focus on allocating resources and scheduling tasks on those resources such that the workflow completes within the given time and cost constraints. The authors in [57] employ empirical heuristics based on assumptions about the workflow and the characteristics of the provisioned resources to propose a scheduling algorithm that dynamically adapts the resource allocations during run-time. Efforts have also been made in dynamically provisioning resources from cloud platforms to extend traditional scientific computing environments such as clusters and batch systems [45, 75, 76].

There have also been efforts that provide resource allocation techniques for a specific class of applications they study. For example, the work in [132] considers a class of applications with flexibility in the quality of the computational results (e.g., accuracy) they provide. The authors develop a model mapping the parameters that determine the quality of the computations to the resource requirements. The model is then applied to determine the resource allocations that meet the budget and time constraints set by the user while maximizing the quality of the computation results. Stewart et al. [105] build a model describing the resources needs of a multi-component application using instrumentation in the operating system.

The work described in this dissertation focuses on determining the resource allocation that achieves efficient operation of large concurrent workloads in any operating environment deployed on the distributed computing systems of clusters, clouds, and grids. This work considers the operating environment to be deployed on resources that are exclusively provisioned and maintained for the operation of an instance of the application. It will present techniques for determining the scale of resources that achieve efficient operation in each exclusive deployment.

2.5 Middleware Techniques

Large-scale, resource-intensive, and scientific applications are built and operated using middleware, such as operating systems [21], batch scheduling systems [49], execution engines [53, 62]. The applications are typically constructed using the programming interfaces provided or imposed by the middleware. This allows the middleware to direct and manage the execution of the application on the available resources.

Several programming frameworks exist for the development and execution of data-intensive workflows and applications [5, 53, 78, 87]. Recently, the use of execution engines for building and running resource- and data-intensive applications have grown in popularity [53, 62, 84]. Hadoop [53] is an open-source implementation that supports the MapReduce programming paradigm [38]. Dryad provides a programming framework and distributed execution engine for DAG-based workloads [62]. CIEL extends the programming and execution model to support dynamic data dependencies and arbitrary data-dependent control flows [84]. These frameworks offer several advanced features in allowing users to modify their workflows to run on distributed systems and improve the run-time performance and time-to-completion. However, most of these frameworks require prior installation on the resources allocated for operation and therefore can be an inconvenience for on-demand deployment and operation of applications.

These frameworks were designed for operation within a single infrastructure or administrative domain and are similar to batch systems [49, 55] in their scheduling and management of resources. However, unlike batch systems, these distributed execution engines also control and manage the workload or data partitioning, data transfer, and locality when executing their applications. Since these decisions are often hidden from the applications, it is difficult for the operators of the applications to accurately estimate and provision the right size of resources required for their execution. This impacts their runtime performance and lowers their cost-efficiency when deployed on cloud platforms.

2.6 Application-level adaptations

Application-level techniques and adaptations have been proposed and studied in several fields such as databases [61], distributed programs [46], and multimedia [115] systems [11]. The work in [9] considers applications that adapt their workload according to the resources available for operation and presents compiler-level techniques for building such applications. The authors in [51] survey techniques, such as partitioning, in the database layer for adapting query plans to changing operating conditions.

Previous efforts have successfully advocated and demonstrated the use of application-level models and techniques for overcoming challenges in various operating environments [32, 44]. The work in [32] advocates application-level scheduling techniques for efficiently running independent jobs on heterogeneous resources in a grid. Doyle et al. [44] formulate and apply an internal model of the behavior of service-oriented programs to predict their memory and storage requirements and provision resources for multiple competing programs running on a shared infrastructure.

This dissertation advocates and applies application-level techniques for achieving cost-efficient deployment and operation of concurrent applications in distributed computing platforms.

CHAPTER 3

ELASTIC APPLICATIONS

3.1 Introduction

The execution environments offered by distributed computing systems can be very challenging since they consist of heterogeneous resources with rapidly changing availability and a high probability of failure. For example, the Spot-Pricing [17] service offered by Amazon provides virtual machines whenever the market price falls below the user's threshold. An application running on this service must be prepared for the addition and removal of resources at a moment's notice. In this way, it is not unlike a grid using a cycle-stealing strategy such as Condor [72] or BOINC [20]. In addition, the execution environments can vary in their behavior and characteristics between invocations of the applications due to differences in the hardware and characteristics of the platforms on which they are deployed.

Previously built scientific applications are not prepared for such execution environments, because they are usually built around the assumption of fixed, homogeneous resources at a single location. For example, message-passing applications are usually designed to run on a fixed number of processors – usually a power of two – that cannot change during runtime. A multi-threaded program is usually designed to run on a fixed number of cores selected at startup; changing the number of cores at runtime results in a serious performance penalty.

Further, these scientific applications lack the ability to harness any currently available resources to proceed with execution and this often leads to poor productivity where valuable time is spent waiting for all requested resources to be available. Also, while fault tolerance can be achieved with checkpointing or other error recovery techniques implemented at the application level, they inherently lack a dynamic fault-tolerance and error-recovery mechanism that will allow for executions to recover from multiple failures, proceed execution or migrate seamlessly to another site in the event of unrecoverable failures. The behavior and performance of such applications vary with hardware, platform, network characteristics [12, 94] and often need to be tuned to suit the platform and hardware used in execution.

In summary, previously built scientific applications are constrained by the availability of dedicated and tightly controlled resources. These applications are inefficient in their utilization of available resources and any resources that become available during run-time. Such limitations prevent these applications from successfully leveraging the scale and affordability of operation on distributed computing systems.

This chapter introduces and defines *elastic applications* as the paradigm of choice in building applications for distributed computing systems. It describes the necessary characteristics that every elastic application must exhibit to achieve successful operation on distributed computing systems. It goes on to present the conversion of a scientific application to an elastic application that exhibits these characteristics. This chapter concludes with an experimental demonstration of the scalability, resource adaptability, fault-tolerance, and portability of elastic applications during their operation.

3.2 Definition and Characteristics

We define applications that adapt their operation to the characteristics of the deployed environment as **elastic applications** [92]. These applications exhibit flexibility in the execution of the workload, tolerance to failures, adaptability to the resources available for operation, and portability across different platforms. They can dynamically adapt to the size, scale, and type of resources available at any given moment, and tolerate failures in the hardware and execution environment. Further, their deployment can be moved to a different platform or infrastructure with minimal effort and intervention from the operators. As a result, elastic applications are well-suited to operate in the dynamic environments of distributed computing systems such as clusters, clouds, and grids.

The characteristics of elastic applications are described as follows:

- **Adaptability.** They must adapt to resource availability during run time. That is, they must dynamically expand their resource consumption to include resources that become available during execution. At the same time, they must also adapt to resources being lost or terminated during execution. These adaptations must occur seamlessly without intervention from the operators.
- **Fault-tolerance.** They must continue execution in the presence of run-time failures. They must isolate failures to individual executions or resources and dynamically recover by re-running the failed executions or by migrating them to successfully operating resources.
- **Portability.** They must be portable across different platforms and environments with limited effort from users. They must be able to execute on any platform using the software components specified by users for that platform without having to be re-engineered or rewritten.
- **Versatility.** By leveraging their portability, elastic applications must be able to simultaneously harness resources with diverse operating environments. In other words, users must be able to run them using resources federated from any cluster, cloud, or grid platform as long as the software compatibility with the different operating environments is established.

These characteristics further enable elastic applications to achieve **scalability** and **reproducibility** without being tied to a specific operating environment or hardware.

3.3 Programming Model and Architecture

In this work, I consider workloads that can be expressed and operated using the *split-map-merge* paradigm and show their construction as elastic applications. This paradigm encompasses the bag-of-tasks [35], bulk synchronous parallel [50],

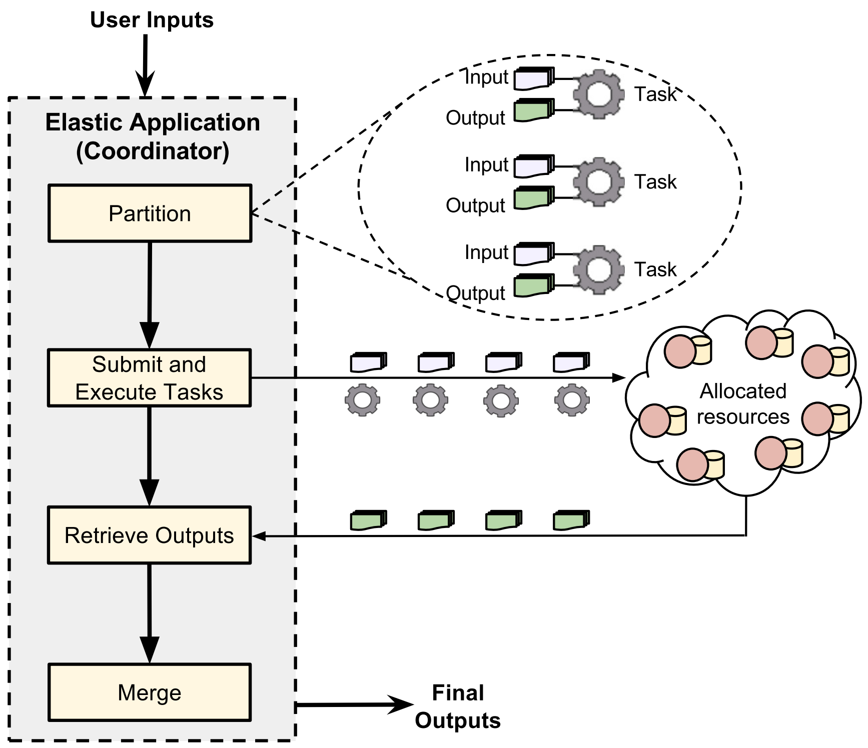


Figure 3.1. Architecture of split-map-merge based elastic applications.

scatter-gather [36], and the popular Map-Reduce [38] (where the split is done during the initial upload of the data to the filesystem of the execution framework) models of concurrent programming. A number of large-scale workloads are expressed and operated using the split-map-merge paradigm [8, 74, 112]. A formal expression for the split-map-merge paradigm is formulated below.

$$Workload : f(N) \rightarrow O, \tag{3.1}$$

$$Split(N, k) : N \rightarrow \{s_1, s_2, \ldots, s_k\}, \tag{3.2}$$

$$Map(k) : f(s_i) \rightarrow O_i \;\; for \; i = 1, 2, \ldots, k, \tag{3.3}$$

$$Merge(k) : \{O_1, O_2, \ldots, O_k\} \rightarrow O. \tag{3.4}$$

Equation 3.1 describes the overall operation of the workload which performs a transformation on input N to produce output O. The split step in Equation 3.2 takes a parameter k and splits N into k partitions. The map step runs the transformation on each of the k partitions as shown in Equation 3.3. Finally, the merge step in Equation 3.4 aggregates the outputs of the individual map functions to produce the final output O. The merge function could be a simple concatenation of the outputs or a sophisticated function (e.g., merge of values in sorted order) depending on the workload. It is important to note the value of O is not affected by the choice of k. The concurrency in split-map-merge workloads results from the simultaneous execution of the map operations.

Figure 3.1 presents a overview of the architecture of elastic applications using the split-map-merge paradigm. The workload of the applications can be expressed in a variety of forms. It could be a directed acyclic graph or *workflow* in which the tasks and their relationship are fully elaborated in advance. It could be a data decomposition in which a large dataset is broken into pieces and processed independently. It could be an iterative algorithm in which a set of tasks is dispatched, evaluated, and then dispatched again until an end condition is met. These structures can describe a variety of applications, such as Monte-carlo simulations [22], protein folding [8], and bioinformatics [112]. Also, the workloads of the elastic applications can consist of a single, multiple, or iterative split-map-merge phases.

Elastic applications are typically implemented by constructing a long-running **coordinator** that submits a large number of short-running **tasks**. The coordinator is responsible for observing available resources, decomposing the workload into tasks of appropriate size, submitting and monitoring tasks, handling fault-tolerance, and interacting with users. The individual tasks are usually self-contained executable programs, along with their expected input and output files. The individual tasks may make use of local physical parallelism in the form of multi-core machines or accelerated hardware such as GPUs, while the coordinator operates at a parallelism of anywhere from one hundred to ten thousand tasks running simultaneously.

3.4 Middleware

The construction and operation of elastic applications on distributed computing environments can consume extensive effort, time, and cost, especially if these applications run large and complex workloads. To lower the level of effort and cost required, developers can take advantage of frameworks and middleware to engineer their applications through the offered API and library interfaces.

A framework chosen for building and operating elastic applications must implement and provide the necessary interfaces for achieving the required properties of resource adaptability, fault-tolerance, portability, versatility, and scalability. In addition, the framework must be able to operate independent of the platform, operating environment, and hardware provisioned for operation. It must also hide the complexities and heterogeneity in the underlying platforms and hardware from the applications. Finally, it must offer easy to use interfaces and facilitate quicker development and deployment of the applications to different platforms and providers with minimal effort from the developers and operators of the applications.

In this work, we use the Work Queue [127] framework developed by our research lab at the University of Notre Dame. It is important to note that are several other middleware and frameworks can be utilized for building elastic applications [53, 62, 65]. We chose Work Queue since it provided us the opportunity to incorporate, study, and evaluate the application and middleware techniques presented in this work.

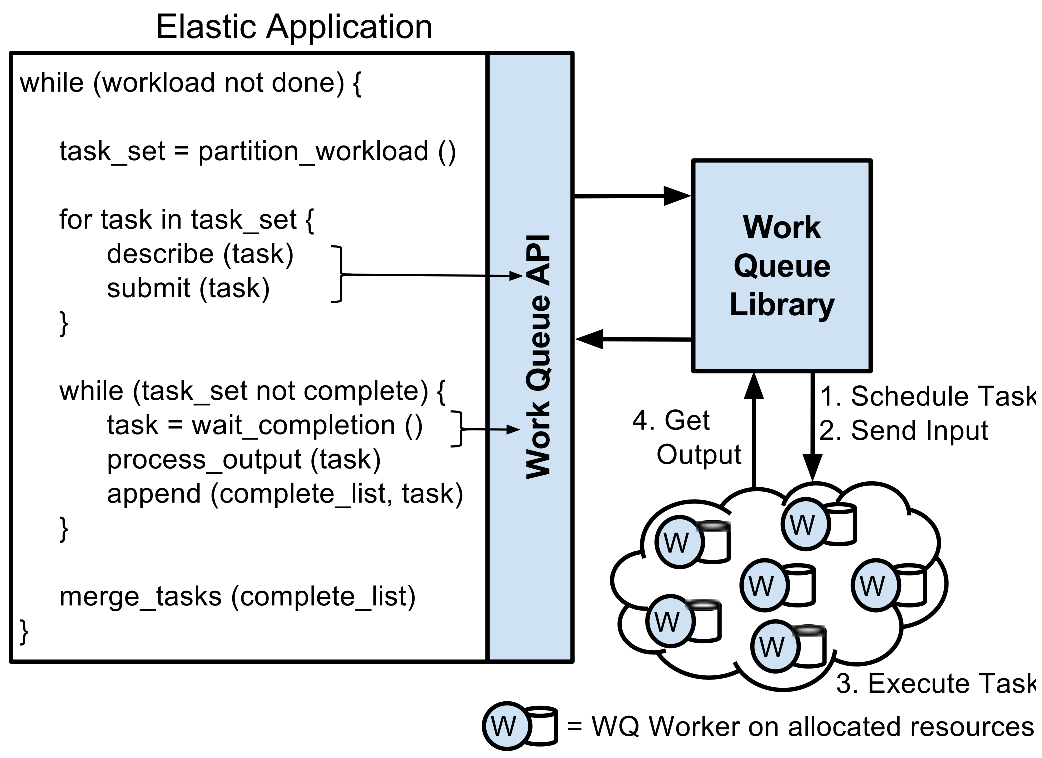
The Work Queue [26] framework is used to implement the master-coordinator of the elastic applications. Work Queue provides interfaces for describing the tasks of a workload, submitting the tasks for execution, and retrieving the results of the executions. Work Queue has been used to build a number of elastic applications in fields such as molecular dynamics, data mining, bioinformatics, and fluid dynamics [8, 83, 92, 93, 112].

The Work Queue framework is based on a master-worker execution model. It consists of the following components:

- **Work Queue Library** implements and provides the functionality for coordinating the workload execution across workers. It schedules tasks on workers, transfers the inputs and outputs of tasks, and reschedules failed tasks. The library also provides data management capabilities, such as caching and scheduling policies that favor workers with pre-existing data.
- **Work Queue API** provides the interfaces (in C, Python, and Perl) to the Work Queue library. The API is used to create master (coordinator) programs that create, describe, and submit tasks for execution, and retrieve their outputs upon execution.
- **Work Queue Worker** is a lightweight process that is run on the allocated resources. It connects to a specified Work Queue master and executes the dispatched tasks.

The master-coordinator of the elastic applications is implemented by their developers using the Work Queue API. The individual units of execution of the application, referred from here on as *tasks*, are partitioned and described by the master. The input files, the executables and execution commands to run, and the output files of the tasks are specified by the master using the API of Work Queue. The underlying Work Queue library dispatches and coordinates the executions of the tasks across the connected workers, aggregates the completed results and returns them to the master-coordinator. Figure 3.2 presents the outline of the construction and execution of an elastic application using Work Queue.

The Work Queue library dispatches tasks to workers as they establish connection and reschedules tasks running on terminated workers. This allows applications to adapt to the size of the available resources during runtime. Work Queue automatically reschedules tasks that failed due to resource failures and allows the application to examine completed tasks and resubmit tasks with erroneous results, thereby enabling



```
while (workload not done) {

    task_set = partition_workload ()

    for task in task_set {
        describe (task)
        submit (task)
    }

    while (task_set not complete) {
        task = wait_completion ()
        process_output (task)
        append (complete_list, task)
    }

    merge_tasks (complete_list)
}
```

Figure 3.2. Construction and operation of applications using Work Queue. The code on the left is an outline of an elastic application.

fault-tolerance in its applications. Work Queue requires applications to explicitly specify the software and data dependencies for the tasks so the environment needed for execution can be created at the workers without concern for the native execution environment. This enables Work Queue to provide data management facilities, such as caching and scheduling policies that favor workers with cached data.

The workers are deployed as executables on the provisioned resources and they are invoked as standalone processes. The workers can be compiled, installed, and run on any POSIX compliant environment. This implies that the worker can virtually be deployed and run on any operating environment including Microsoft Windows based environments (using Cygwin [1]).

3.5 Example: Elastic Replica Exchange

We construct an example elastic application to experimentally demonstrate and evaluate the characteristics of elastic applications and the benefits they present for operation on distributed computing systems. In this chapter, we consider the construction of replica exchange molecular dynamics simulation as an elastic application. Replica exchange molecular dynamics [25, 107] is a technique used to improve the sampling of the potential energy surface in a protein system. Further details about the replica exchange algorithm and its implementation as an elastic application are presented in Chapter 4.

The replica exchange computations are typically implemented using MPI. Some of the simulation software such as ProtoMol [77] and Gromacs [71] offer built-in MPI implementations for this purpose. However, as we discussed earlier in Section 3.1, the MPI based implementations exhibit disadvantages and inefficiencies when operated on distributed computing systems. So to overcome the shortcomings of MPI based implementations of replica exchange, we built an elastic implementation using the Work Queue framework described above.

The implementation of elastic replica exchange involves the creation of the master (coordinator) script described in Section 3.3. The master is built using the Work Queue API. The master creates and specifies the configuration and input files required for each iteration of the computation and gathers the output files upon their completion by the Work Queue workers. At the end of each iteration, the master checks to see if an exchange can be attempted between two replicas and if so swaps the necessary parameters of those replicas. The master then proceeds to generate the configuration and input files for the next iteration. The master, therefore, coordinates the entire simulation across the workers running on the allocated resources for operation. The tasks created by the master use the ProtoMol [77] as the computation kernel for running the simulations on their inputs.

In the next sections, we experimentally study and evaluate this elastic implementation of replica exchange.

3.6 Comparison against an Existing Construction using MPI

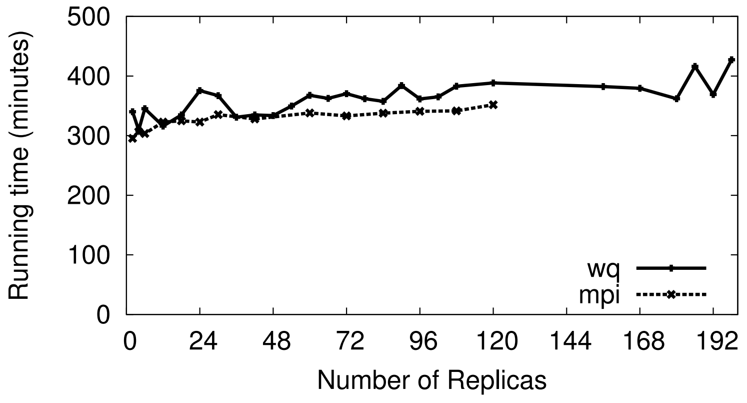


Figure 3.3. Running time performance comparison of MPI- and Work Queue-based implementations of replica exchange.

We first compare the performance of the elastic implementation of replica exchange using Work Queue against its MPI implementation. For the experiments in this section, we deployed and ran both implementations on the Sun Grid Engine [49] infrastructure at the University of Notre Dame. Each experimental run involved simulations over 100 Monte Carlo steps with each step running 10000 molecular dynamics steps. Figure 3.3 compares the running time of the MPI- and Work Queue-based implementations of replica exchange. The number of workers deployed and run was equal to the number of replicas simulated in the experiment. For example, a run with 30 replicas had 30 workers being deployed and run. The running time of these experiments were measured from the start of simulation to its completion. Therefore, Figure 3.3 does not include the job queuing and scheduling delays. In this figure, we observe that the Work Queue implementation has a slightly higher running time than the MPI implementation. This is attributed to the communication and data transfer overheads between the master and workers running remotely.

An important observation we make in Figure 3.3 is that the MPI runs do not scale well beyond 120 replicas. This is due to the lack of resources at this scale being available simultaneously on the Notre Dame SGE infrastructure. We also note that the Notre Dame SGE has a constraint on the resources available to a user at any given time, thereby limiting the scalability of the MPI implementation even with the availability of more resources. On the other hand, we see that the Work Queue implementation scales well due to its ability to scavenge and utilize resources as they become available. Experiments beyond 120 replicas were achieved by deploying multiple workers on resources with multiple computing cores and invoking their execution on each core through MPI. This illustrates better scalability characteristics of the Work Queue implementation of elastic replica exchange.

3.7 Experimental Illustration of Characteristics

In this section, we evaluate the operation of elastic replica exchange in terms of the characteristics identified in Section 3.2. We begin by describing the platforms used in the experiments presented in this section.

Campus SGE. The Sun Grid Engine (SGE) at the University of Notre Dame is a dedicated platform for running high performance scientific applications. The jobs are submitted to the compute nodes via the SGE batch submission system [49]. The compute nodes run Red Hat Enterprise Linux (RHEL) as their operating environment. The compute nodes here are typically composed of high-end hardware. The

TABLE 3.1

PLATFORMS USED IN THE EXPERIMENTAL STUDY AND DEMONSTRATION OF ELASTIC REPLICA EXCHANGE

Name	System	Processor	I/O
Platform A	Amazon EC2	2*2 x 1.0-1.2 GHz	7.5 GB memory
Platform B	Notre Dame SGE	2*2 x 2.6 GHz	8-12 GB memory
Platform C	Microsoft Azure	2 x 1.6 GHz	3.5 GB memory

workers were queued and submitted as jobs to this grid. Upon being scheduled and run, the workers connect to the master and execute the assigned workloads.

Amazon EC2. The Elastic Compute Cloud or EC2, built by Amazon.com, is a platform that allows virtual machine instances to be requested, allocated, and deployed on demand by users. Different instance sizes are provided with varying hardware configurations to satisfy different requirements and workloads of the users [18, 89]. The instances allocated can be installed and run with different Linux operating system flavors and kernels and their operating environments can also be customized. Since the instances can be installed and customized to run a Linux environment, the migration of our implementation to EC2 was similar to SGE.

Microsoft Azure. The Windows Azure platform, from Microsoft, offers virtual instances running an image of the Azure operating system. The virtualized instance is offered through the Azure hypervisor and provides an operating environment based off the Windows Server 2008 R2 VM system [58, 80]. There are two computational roles offered in the platform - the web role that serves as the front end interface to the allocated compute instances, and the worker role that serves as the core computing unit that runs tasks and applications. As a result of this two tiered architecture, we built wrapper scripts that communicate to the web role and invoke workers on the worker roles. We also used Cygwin-compiled executables in migrating our implementation to this environment.

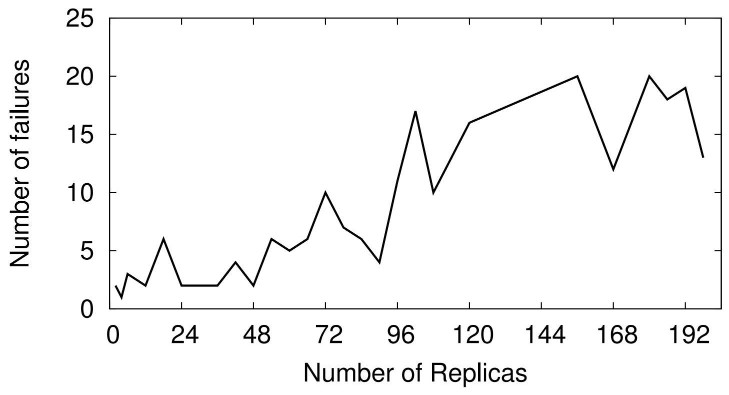


Figure 3.4. Failures observed with Replica Exchange using Work Queue.

The failures include both application-level (such as failure to reach desired simulation state) and system-level failures (such as timeouts on data transfer).

Figure 3.4 illustrates the number of failures that occurred on the worker sites when running on the campus SGE platform. These failures are attributable to a variety of factors such as stalled workers, application faults, data transmission failures,etc. With the MPI-based implementation, these failures will stall the entire experiment and will need to be restarted and rerun. The Work Queue implementation offers fault-tolerance by rerunning only the failed task in the experimental run, and in the event of any unrecoverable failures at a worker, migrating its execution to a different worker.

We now proceed to port and study the elastic implementation of replica exchange on two cloud computing platforms, Amazon EC2 and Microsoft Azure. We study and demonstrate the behavior of our elastic implementation on these different platforms. Our objective is *not* to compare the performance of the distributed computing platforms against each other, as they offer different cost-performance trade-offs using different hardware. Instead, our goal here is to show that our system functions correctly and portably across multiple different environments.

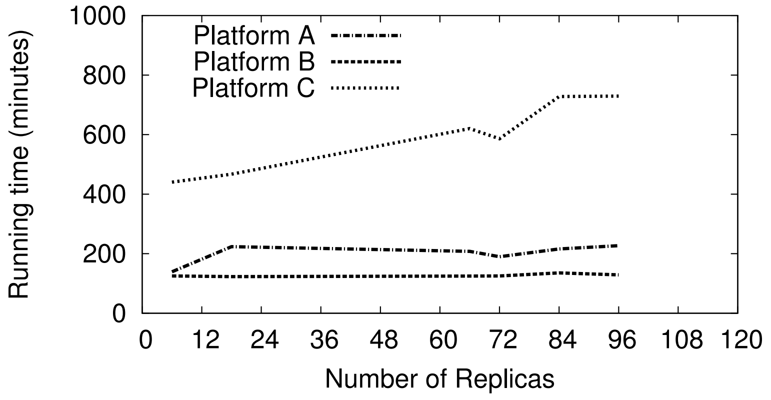


Figure 3.5. Comparison of running times on the platforms described in Table 1.

In the experiments described below, the number of workers deployed were again equivalent to the number of replicas involved in the simulation run. Each experiment ran simulations performed over 100 Monte Carlo steps involving 10000 molecular dynamics steps each. Figure 3.5 shows the running times of experimental runs with varying replica sizes on these platforms. The x-axis represents the number of replicas involved in each run. Figures 3.6 and 3.7 plot statistical data on the completion time of the individual Monte Carlo steps on the three platforms from experimental runs involving 18 replicas. Figure 3.6 plots the cumulative distribution function of the

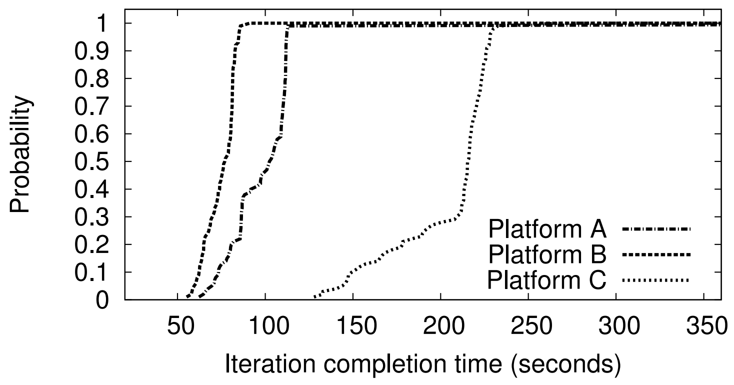


Figure 3.6. Cumulative distribution function of the completion times of the iterations in Replica Exchange using 18 replicas.

completion time of each step. Figure 3.7 gives the corresponding histogram plotting the distribution of completion times after being classified in bins.

From Figures 3.6 and 3.7, we notice significant variations in the completion times of the Monte Carlo steps on one of the platforms (Platform C) as compared to the other two platforms. While these variations can be attributed to one or more of several factors including network latencies and jitter, virtualization effects, firewall, load balancing etc., this is good evidence that our implementation is impervious to any peculiar platform and network characteristics of a distributed computing system. Our implementation demonstrates the ability to hide differences in the characteristics and behavior of different distributed computing platforms from the application.

We also demonstrate the execution of elastic replica exchange across multiple distributed computing platforms. We show this by deploying workers across all three platforms in Table 1. The experimental run involved 75 replicas simulated over 100 Monte Carlo steps running 10000 molecular dynamics steps. Figure 3.8 presents the number of workers connected, the completion time of each Monte Carlo step, and the running cost during this experimental run. The experiment was started with 25 workers being submitted and run on the Platform B. Since there are 75 tasks, as a

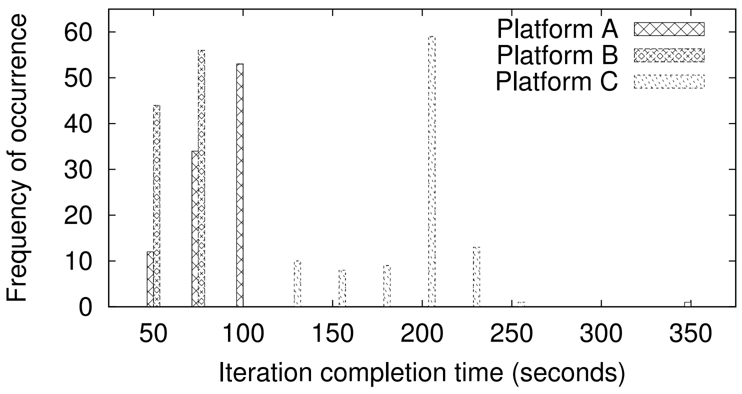


Figure 3.7. Histogram of the completion times of the iterations in Replica Exchange using 18 replicas.

The bins used in the plot were of size 25 time units (seconds) and each bin consists of all values that are greater than or equal to the corresponding bin label. The bin labels are plotted on the x-axis.

task corresponds to a simulation step of one replica, it takes three round-trips of task execution for these workers to finish a Monte Carlo step.

After an hour (around 4200 seconds in Figure 3.8), we deploy and run 25 workers on instances in Platform A bringing the total number of workers to 50. This addition of resources lowers the completion time of Monte Carlo steps as it requires only two round-trips of task executions across the workers. We then deploy and run another 25 workers on Platform C after two hours of run-time (around 7600 seconds). We immediately observe a spike in the running time which we attribute to the long transfer times in sending the simulation program, ProtoMol, to the added workers. We also observe from Figure 3.8 that the addition of these workers on Platform C results in an increase in the running time of each step. This is because the running time of simulation steps on Platform C significantly varies from the other platforms as we observe in Figure 3.5. We attribute these to differences in the hardware, network, and system characteristics and specifications of these platforms. As a result, adding workers on Platform C negates any benefits gained from the parallelism in running

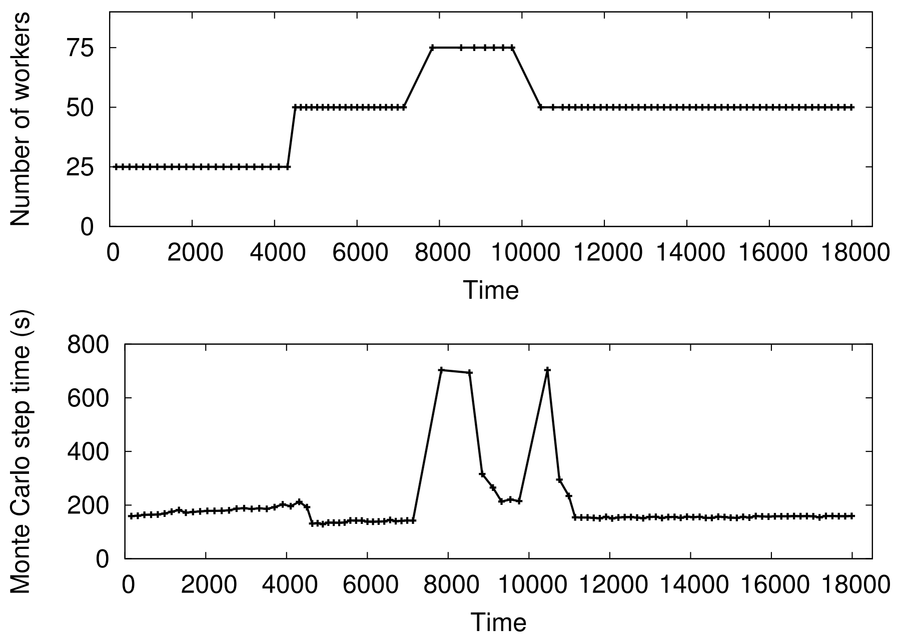


Figure 3.8. Illustration of a run across all three platforms described in Table 1.

75 tasks simultaneously. We manually removed the workers on Platform C after an hour of their deployment (around 10100 seconds) to speed up the completion of the experiment. This results in the failure of tasks running on the removed workers. The spike in the running time following the removal of these workers is attributed to the failed tasks being scheduled and rerun on the remaining workers.

We continue with an evaluation of the work queue implementation of replica exchange focused primarily on its ability to adapt to resource availability. In this experiment, we run a replica exchange simulation with 400 replicas over 100 Monte Carlo steps. Figure 3.9 plots the time to complete the simulation of a Monte Carlo step over all replicas. The master script was run from a workstation inside the Notre Dame campus network. We note that the master can be run from any site including distributed computing platforms, but we chose this setup to illustrate ease of building and deploying the master.

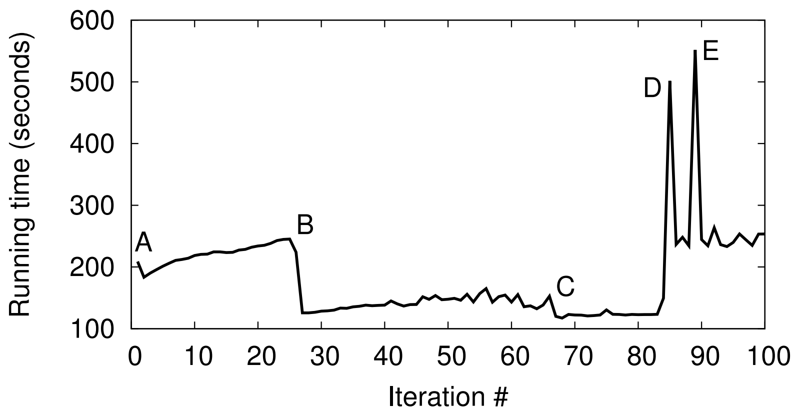


Figure 3.9. Running time of Monte Carlo steps run over 400 replicas with workers running on multiple distributed computing systems.

TABLE 3.2

DESCRIPTION OF THE EVENTS SHOWN IN FIGURE 3.9

Event	Description	Total workers after event
A	Start of experiment with 100 workers in ND SGE	100
B	Addition of 150 workers in Condor	250
C	Addition of 110 workers in Condor and 40 workers in Amazon EC2	400
D	Removal of 100 workers in ND SGE	300
E	Removal of 125 workers in Condor and 25 workers in Amazon EC2	150

We utilize workers on three different platforms: our 4000-core campus cluster managed by SGE, our 1200-core campus grid managed by Condor, and the Amazon EC2 service. Over the course of the experiment, the computing resources varied dynamically as compute nodes were requested, allocated, and terminated. The specific instances at which the available resources changed is labeled in the figure and described in Table 3.2.

We make the following observations from Figure 3.9. The work queue implementation of replica exchange is elastic in dynamically adapting to resource availability. We observe this at each of the described events in Table 3.2. At Events B and C, where the addition of workers to the existing pool results in the running time of the Monte Carlo steps being lowered. At Events D and E, the termination and removal of workers only leads to an increase in the running time without stalling the simulation run. We also observe that the work queue implementation is fault tolerant and recovers from failures. We observe this at Events D and E, where workers where removed while they were executing tasks corresponding to the Monte Carlo simulations of the replicas. This resulted in the failure of tasks being executed by the removed workers. Work Queue dynamically rescheduled these failed tasks on the remaining workers for handling failures resulting from tasks being killed or terminated. We attribute the spike in the running time following Events D and E to the failed work units being rerun on the remaining workers as they finish execution of their assigned tasks.

Finally, we make the observation that the simulations completed with individual tasks executing at resources provisioned from different distributed computing platforms. The Work Queue implementation of replica exchange is oblivious to the underlying execution platform and environment and thus enables multiple cloud platforms to be leveraged at the same time. This is especially useful, when the user is constrained by policies on the number of running processes inside a cloud platform, access to limited set of resources in commercial cloud platforms due to monetary constraints, etc. In summary, the evaluations performed on different distributed computing platforms in Figures 3.8 and 3.9 show the portability of the Work Queue-based implementation of replica exchange.

3.8 Conclusion

This chapter showed that elastic applications are well-suited for operation on distributed computing systems by their ability to adapt to resource availability, recover from failures, scale, and portability across multiple execution environments. The adaptability of elastic applications to the available resources allows operators to scale up and down the resource allocations based on their evolving needs and cost constraints. Their fault-tolerance enables progress in execution even in the presence of multiple simultaneous failures. The platform-independent operation and the portability of these applications allows operators to easily migrate or extend their execution to a different platform or infrastructure of choice without having to redesign or rewrite the application. Further, these characteristics enable elastic applications to achieve scalability in a single platform or across multiple platforms through federation while eliminating the need for dedicated and sophisticated hardware.

This chapter also showed the elastic implementation of replica exchange can be slower compared to its MPI-based implementation. A contributing factor is the overheads of the transfer of the inputs and outputs between the master and its workers and the use of a global synchronization barrier. In the next chapter, we analyze the construction and operation of the elastic replica exchange application and a few other applications used in different scientific fields. The analysis is used to identify the bottlenecks in operation and build techniques to eliminate those bottlenecks.

CHAPTER 4

CASE STUDIES OF ELASTIC APPLICATIONS

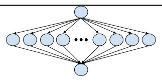
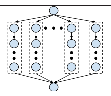
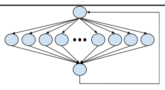
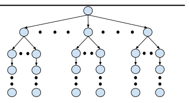
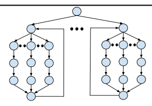
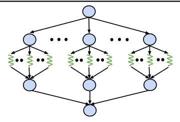
4.1 Introduction

This chapter presents experiences and observations in designing and implementing a selection of elastic applications on clusters, clouds, and grids: Elastic Sort (E-SORT), Elastic MAKER (E-MAKER), Elastic Replica Exchange (REPEX), Folding At Work (FAW), Accelerated Weighted Ensemble (AWE), and Elastic Bowtie (E-BOWTIE). Table 4.1 summarizes the properties of these applications. This chapter also describes the challenges encountered in the construction and operation of these elastic applications in heterogeneous and dynamically changing environments. It presents and explains the techniques applied in each application to overcome the challenges. The chapter concludes by establishing six broad guidelines derived from the presented techniques and other observations from building and operating these applications: *(1) Abolish shared writes, (2) Keep your software close and your dependencies closer, (3) Synchronize two, you make company; synchronize three, you make a crowd, (4) Make tasks of a feather flock together, (5) Seek simplicity, and gain power, and (6) Build a model before scaling new heights.*

The applications in this chapter are built using the Work Queue [26] framework, but the guidelines presented in this chapter apply to many other elastic applications constructed using other programming frameworks.

TABLE 4.1

PROFILE OF THE STUDIED ELASTIC SCIENTIFIC APPLICATIONS

Application	Function	Input	Kernel	Work Queue Code Size	Logical Structure
E-Sort	Data processing	File containing integer records	GNU Sort	∼ 750 lines	
E-MAKER	Annotate genome sequences	Anopheles gambiae genome	MAKER	∼ 1150 lines	
REPEX	Sample conformational space of proteins	WW protein domain	ProtoMol	∼ 700 lines	
FAW	Study of protein dynamics	Alanine Dipeptide molecule	Gromacs	∼ 600 lines	
AWE	Protein folding	WW protein domain	Gromacs	∼ 1000 lines	
Bowtie	Genome alignment	Culex quinquefasciatus genome	Bowtie2	∼ 250 lines	

4.2 Elastic Sort (E-SORT)

Operations on large sizes of data has traditionally been one of the driving factors behind the push for greater compute capacity and storage. These operations typically involve processing, analyzing, or mining for patterns in data. In this dissertation, I consider the sorting of records in a file as a representative workload of many data processing and analysis applications.

Function. The sorting of large datasets can be accomplished within a reasonable time duration by partitioning the dataset, sorting the partitions individually, and merging them in sorted order. This approach is very similar to the merge sort algorithm. Elastic Sort or E-Sort, built using Work Queue, provides an implementation for the sorting of large data sets using resources in distributed computing systems. E-Sort partitions the N data records and dispatches each partition to a remote worker instance where it is sorted. The sorted partitions are then aggregated and merged at the master-coordinator to produce the final sorted sequence. The sorted partitions are merged using a simple k-way merge. The merge algorithm iteratively compares the records in the K partitions and selects them in sorted order to produce the final output. The asymptotic running time of the algorithm is $O(N * K)$. It is important to note that the goal here is not to build an optimized implementation of the sort algorithm but to demonstrate and study the elastic application. The logical structure of E-SORT is illustrated in Figure 4.1.

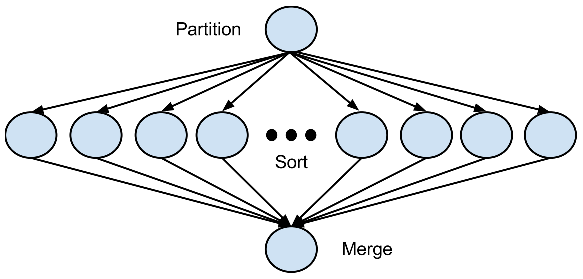


Figure 4.1. Logical structure of E-SORT.

Construction and scale of operation. The implementation of E-Sort using Work Queue consists of about 750 lines of code in the C programming language. E-Sort has been successfully used to sort over 20 billion integer records totalling 111 GB in size. To limit long execution and wait times for results, I perform experiments using a smaller data size consisting of 2 billion integer records that total 11 GB in size. When this data was sorted sequentially using the GNU Sort kernel, it took more than 2 hours to complete on a 64-bit machine with 12 GB of physical memory. In contrast, E-Sort using carefully determined operating parameters sorted this data in less than 50 minutes using 18 cores.

The distributed sorting of records using E-Sort is also a good example of a workload that stresses the compute, I/O, and networking subsystems during operation. The sorting operation is I/O bound since the integer records are read from the file stored in disk and loaded into memory. The sorting of the records in memory involves the execution of arithmetic comparison instructions at the CPU. The sorted values are then written back to disk and transferred to the master where they are merged to produce the sorted sequence. The transfer of the partitioned inputs and outputs to and from the master happen over the network, and therefore the operation of E-Sort also heavily utilizes the network resources.

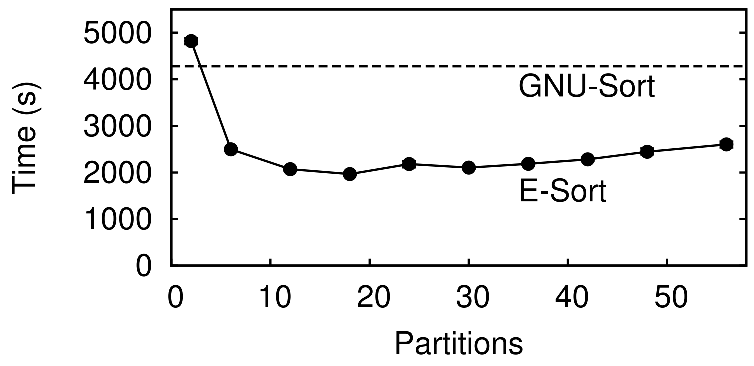


Figure 4.2. Running time of distributed sorting (E-Sort) and sequential sorting.

The experiments were performed on medium-size instances in Microsoft Azure.

Observations. The number of partitions created by E-Sort during operation determines the runtime performance and operating costs. Figure 4.2 presents the running time of E-Sort with different partitions when sorting 2 billion integer records. In this figure, we observe the presence of an optimal partition size at 18 partitions that achieves the fastest running time. Further, the figure shows that a poorly chosen partition size can result in a running time that is worse than the sequential execution of the GNU Sort kernel in sorting the same data. These observations illustrate the need for the careful determination of the number of partitions in each operation of E-Sort. We encounter this challenge in all the elastic applications studied in this chapter.

To correctly determine the partitions for sorting the specified inputs in the deployed operating environment, we need to know the trade-offs between the gains and overheads of operating with a certain number of partitions. This requires a model that formulates the overheads of operation in the deployed environment. The partitioning can then be effectively determined using estimates of the runtime performance obtained from the model for the current operating environment.

4.3 Elastic Replica Exchange

Protein folding is a grand challenge and has been simulated using molecular dynamics (MD), which numerically integrates Newton's equations of motion for all the atoms in a protein system. The potential energy of the protein system dictates the probability of remaining in a given geometric configuration, while the temperature provides energy to jump over barriers in the potential energy surface. Due to the high dimensionality of the problem, MD simulations often get trapped in local minima of the potential energy surface. One way of overcoming energy barriers is to raise the temperature of the system; however, the paths obtained from high temperature simulations do not correspond to the paths at the lower temperature.

A technique used to improve the sampling of the potential energy surface is parallel tempering, also called replica exchange molecular dynamics [25, 107], which has replicas at many temperatures. Many configurations are visited by the high temperature replicas and then annealed to lower temperature replicas by a Monte Carlo procedure that achieves the correct statistical distribution.

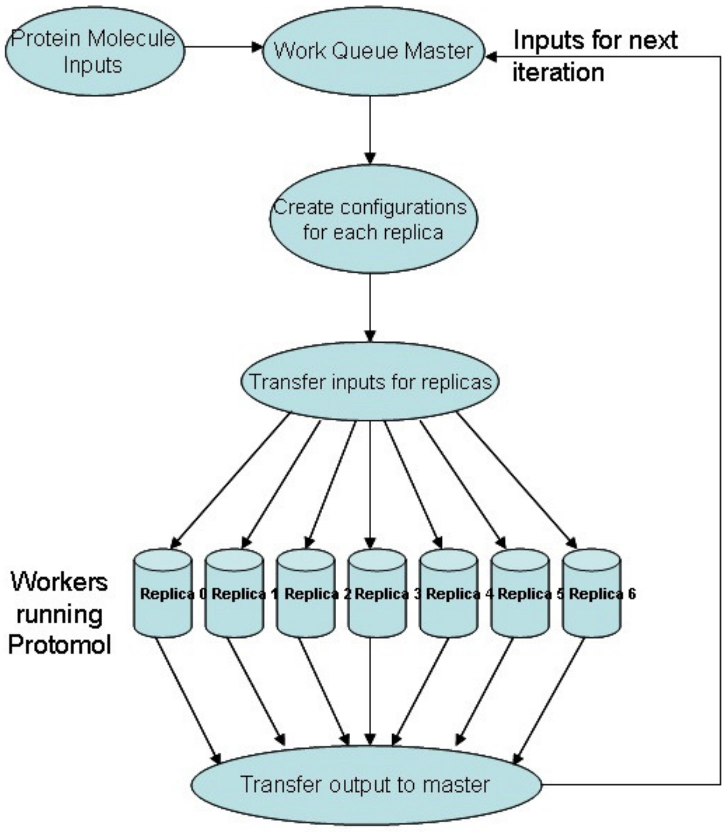
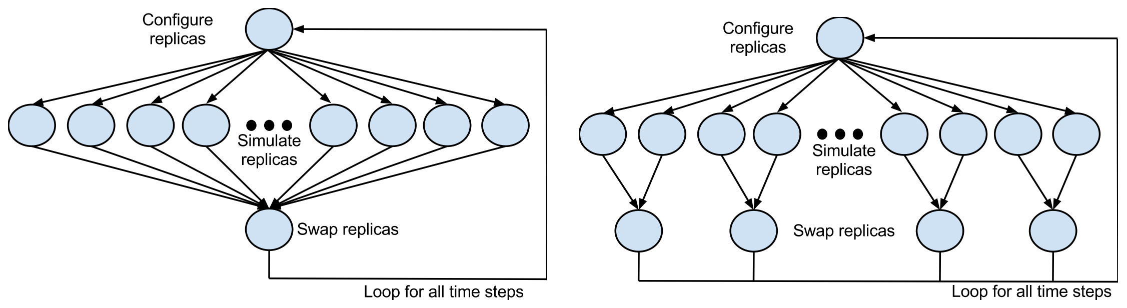


Figure 4.3. Elastic implementation of Replica Exchange using Work Queue.

Function. Replica exchange molecular dynamics simulations are run by creating multiple replicas of a protein molecule and executing each over several Monte Carlo steps or iterations at different temperatures. These replicas are independent of each other and therefore, can be simulated concurrently. At the end of every iteration, an exchange is attempted between neighboring replicas, where if certain criteria are met, the replicas are swapped with regards to their temperature and the simulation is continued. The simulations of replicas in each iteration are completely independent and can be performed parallel to each other. The communication between replicas only happens at the end of each iteration when an exchange is attempted.

Construction and Operation. The previous chapter described the construction of the replica exchange simulations as an elastic application using Work Queue.



(a) Global synchronization (b) Localized synchronization

Figure 4.4. Logical Structure of REPEX.

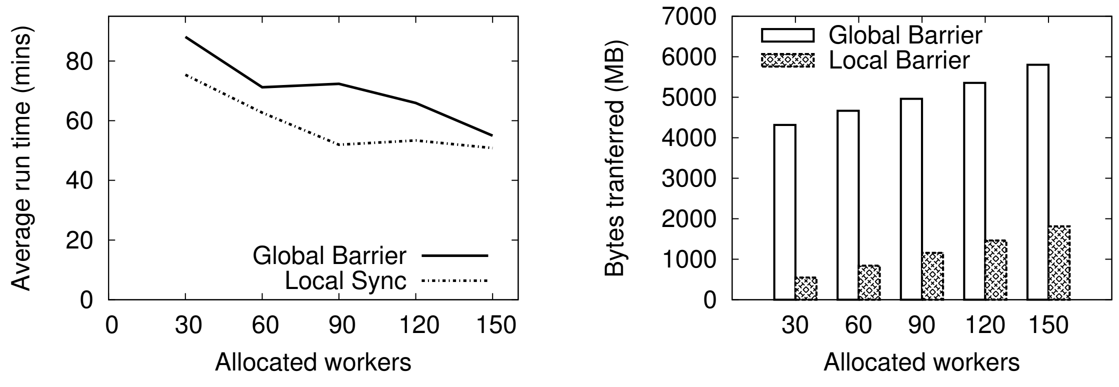
The architecture of the elastic implementation of replica exchange is outlined in Figure 4.3.

Similar to the native MPI implementation, elastic replica exchange was implemented using global synchronization barriers. That is, the entire set of simulated replicas were synchronized at the end of each time step when an exchange is attempted. Each time step, therefore, served as a global barrier. These barriers were used to ensure the two random neighboring replicas chosen to attempt an exchange were at the same time step in the simulation. Figure 4.4a illustrates the logical structure of replica exchange with global barriers.

These global barriers were needed in ensuring correctness of the replica exchange in parallel computing environments to ensure the replicas were in lockstep. Due to the homogeneity in these environments, the performance impact of using each time step as a global barrier was minimal and often overlooked.

Observations. In the heterogeneous operating environments of distributed computing systems, global barriers introduce delays and overheads which adversely impact the overall performance of the application.

Therefore, to run replica exchange simulations efficiently as elastic applications, the presence of global barriers must be removed. So, we replace the global barrier at each time step with barriers spanning two neighboring replicas. We pre-compute the pairs of replicas used for an exchange attempt at each step, and require only those pairs to synchronize at their exchange step. That is, the barriers span a replica pair and occur at the exchange step of that pair. Figure 4.4b illustrates the replica exchange logical structure with localized barriers.



(a) Comparison of the average run times from 10 runs simulating 150 replicas over 150 time steps.

(b) Comparison of the transfer overheads from a single run simulating 150 replicas over 150 time steps.

Figure 4.5. Comparison of the running time and transfer overheads of the global and localized barrier version of REPEX.

The use of local barriers resulted in improvements in the run-time performance over the globally synchronized version as illustrated in Figure 4.5. Figure 4.5a plots the running time of experiments involving 150 replicas averaged over 10 runs. It shows the average running time as the number of workers allocated on the Condor grid at Notre Dame is varied. We observe that the running times of the local barrier version were faster in the presence of shared resources, heterogeneous hardware, resource failures (these factors are also the reasons behind the uneven run times observed for both versions).

The use of localized barriers also enabled the workload to be partitioned such that the tasks within a partition share data and generate outputs that serve as inputs for other tasks in that partition. This avoids the transfer of input and output files at every time step and makes better use of the available storage capacities at allocated resources. Figure 4.5b illustrates the lower transfer overheads of the localized barrier version compared to the global barrier version.

4.4 Genome Annotation (E-MAKER)

Genomic annotation is the process of identifying various cellular entities, such as genes, exons, mRNA, in the genome of an organism. Additionally, genomic annotation seeks to assign functional information to these components by assessing similarity to known genomic components of other organisms. Genomic annotation often utilizes genome prediction as well as comparison against a reference genome to find similarities for use in annotation of the input genome.

Function. MAKER [60] is a commonly used toolchain for genomic annotation. The genomes processed by MAKER are comprised of *contigs*, or large contiguous sequences in a genome. MAKER operates by running a series of tools that process and annotate the input genome. It produces the final output by implementing a consolidation stage at the end that aggregates the outputs of the various tools and extracts the annotations on which consensus is observed. MAKER can be run either sequentially or in parallel using MPI.

Construction and Operation. Elastic MAKER or E-MAKER is an elastic implementation of the genome annotation process that is built using Work Queue. It partitions and dispatches the genome sequences for concurrent annotation on allocated resources. It uses the MAKER tool [60] as the kernel for annotating the partitioned sequences. The detailed construction of the Work Queue based E-MAKER is provided in [113]. Figure 4.6 describes the logical structure of E-MAKER.

Observations. E-MAKER inherits most of the techniques and mechanisms in the native MAKER implementation. However, the native MAKER implementation

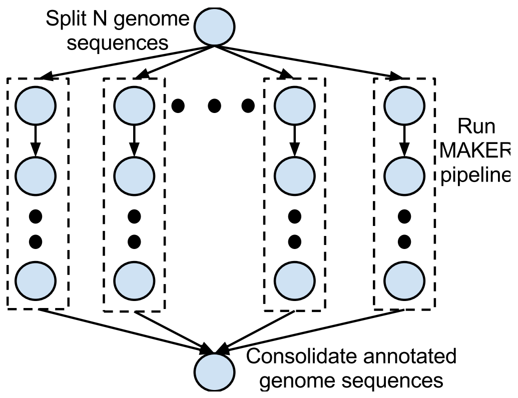


Figure 4.6. Logical Structure of E-MAKER

used MPI to operate in parallel. As a result, its design and development was heavily influenced by the execution environment in parallel computing systems that is homogeneous and consistently available across multiple resources.

For instance, MAKER writes the outputs of its tasks to a common shared file. This allowed outputs to be continuously aggregated in a single file. However, this setup incurs the overheads of file locking mechanisms that enable concurrent and consistent write accesses. Further, MAKER requires a shared filesystems to store and manage its inputs and outputs. Such techniques and requirements limited the ability and performance of E-MAKER on heterogeneous resources.

To improve the performance of E-MAKER, we made the following modifications. First, we modified the tasks to write their outputs in their local execution environment. On completion of the tasks, we gathered these outputs from their execution sites to produce the final output. Second, we explicitly specify the inputs and outputs for tasks to be transfered to and from the resources chosen for task executions. This eliminated the assumption or requirement of a shared filesystem spanning the allocated resources, which is impractical when resources are derived from multiple platforms. In addition, such operating environments include resources that fail or are terminated during run-time. Hence, dedicated and distributed write accesses avoid scenarios where resource failures corrupt the data stored in a shared filesystem. Figure 4.7 compares the failures observed in the E-MAKER implementations using shared file system and dedicated accesses when running on the Condor grid at Notre Dame. The failures in Figure 4.7a are due to (1) failures in write accesses due to locks and (2) failures due to Condor terminating jobs. The dedicated access version of E-MAKER eliminates the failures in write accesses thereby lowering overall failures and improving stability.

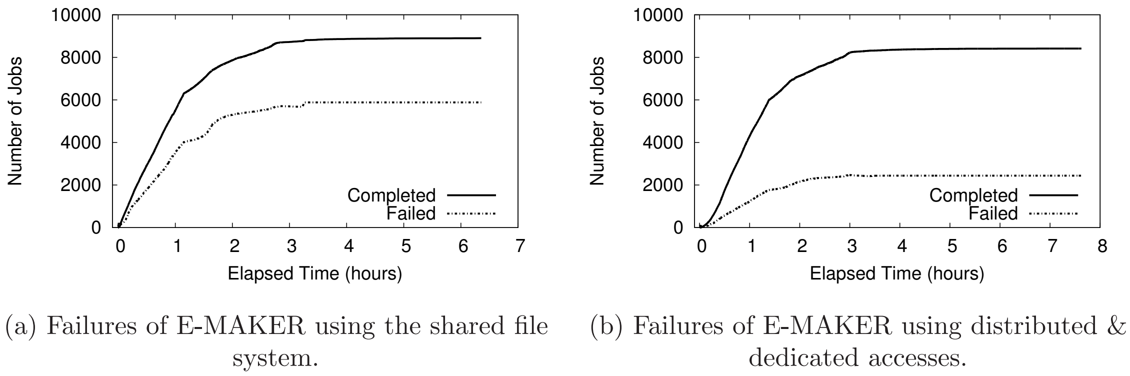


Figure 4.7. Comparison of failures in the E-MAKER versions using shared file system and dedicated data access.

Figure 4.7b shows the number of failures to be lower during run time. We also notice that, despite the lower failure overheads, the overall completion time is longer compared to Figure 4.7a. From our investigation, we attribute this to two factors: (1) overheads from explicitly transferring input and output data, and (2) use of shared and heterogeneous resources. In studying the transfer overheads, we found E-MAKER to operate by simply creating a task for each contig in the input set. This manner of uninformed decomposition often resulted in high transfer overheads leading to sub-optimal run-time performance.

Finally, the native implementation of MAKER utilized a global logging and failure recovery mechanism. This mechanism restarted MAKER at the last successfully logged global state on encountering a failure. We modified E-MAKER to use the failure recovery mechanisms offered in Work Queue. This allowed E-MAKER to isolate failures to individual tasks or resources, migrate tasks from failed resources, validate task outputs, and resubmit tasks with erroneous outputs. With this modification, E-MAKER eliminates the overheads and costs of global logging and recovery in the presence of failures.

We also noticed that the software executables of the tasks in E-MAKER required libraries, such as BioPerl, for their execution. Since the operating environments of the allocated resources are diverse and are not guaranteed to include these software dependencies, we explicitly specify the executables along with their required libraries as the inputs of each task. This allowed the operating environment for each task to be transferred and correctly setup at the allocated resources. This modification enabled E-MAKER to (1) harness resources irrespective of the suitability of their native operating environment to task executions, and (2) handle heterogeneous operating environments by transferring the version of the software components compatible with those environments.

4.5 Folding@Work

A number of biologically significant events in proteins happen at very small timescales (micro-seconds to femto-seconds). To capture these events at such small resolutions, the sampling of these systems has to happen at unprecedented speeds and scales. This requires the use of sophisticated or supercomputing hardware and therefore is constrained by costs and access to these resources. As a result of this limitation, biomolecular scientists have developed alternative approaches that run on commodity hardware. These approaches leverage parallelism in the analysis and simulations techniques of such systems. Specifically, these approaches decompose long trajectories in the protein folding studies into smaller and parallel trajectories that are close approximations.

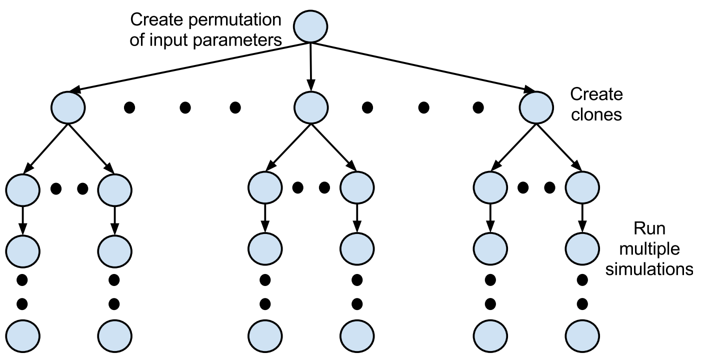


Figure 4.8. Logical Structure of FAW.

Function. The Folding@home project is one such framework that applies parallelization to capture the protein folding phenomenon by running on idle commodity hardware [102]. We build on the Folding@home project to create a framework, called Folding@Work or FAW, that is customizable and flexible to suit the needs of the users and their studies. In FAW, we allow users to specify and customize the simulation environments and the nature of their analysis. We also allow resources from multiple sources and platforms such as Amazon EC2, Microsoft Azure, Condor, etc., to be federated by users to achieve the scale and performance they desire in their experiments.

Construction and Operation. The FAW framework is built using Work Queue to run as an elastic application. This framework is invoked with the specification of

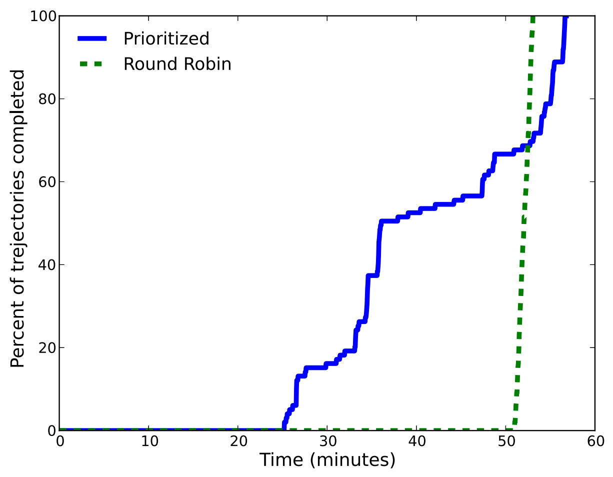


Figure 4.9. Plot comparing the throughput of the prioritized and round-robin approaches in FAW.

several experimental parameters such as molecular structure, temperature, etc. FAW applies these specifications to construct and decompose its workflow into tasks. Our work in [26] describes in detail the construction of FAW using Work Queue. In this paper, we focus on studying and improving the performance of the FAW framework.

Observations. We begin by observing that a typical FAW workflow involves a collection of clones called *trajectories* that are simulated in parallel. Each trajectory represents the path taken by a protein molecule in achieving a folded state. A fully formed trajectory contains the path taken by a protein molecule to reach folded state. The goal of such workflows is to aggregate and study as many fully formed trajectories as possible. As a result, the number of fully formed trajectories dictates the throughput of an experimental run in FAW.

We now describe the technique used to improve the performance of FAW in terms of its throughput. FAW decomposes its workflow into a set of tasks corresponding to each trajectory in the simulations. It uses a round-robin approach in creating and submitting tasks for each trajectory. By replacing the round-robin approach with a mechanism that clusters and prioritizes tasks corresponding to a trajectory closer to achieving fully folded state, the throughput of FAW can be enhanced. Using this insight, we modified the design of FAW to cluster tasks such that trajectories closer to completion are prioritized during execution. The benefit of this approach is illustrated in Figure 4.9 where we compare the throughput, which is the percentage of completed trajectories, with and without prioritization. The experiments in this figure involved 100 clones each running 20 simulations using 50 workers on the Notre Dame SGE cluster. We observe that the clustering and prioritization approach yields fully folded trajectories throughout its run time. This provides opportunities to analyze and gather scientific data much earlier in the runs. This also implies that users can quickly achieve scientific output in running FAW, even with smaller or shorter resource allocations.

4.6 AWE

Function. Accelerated Weighted Ensemble (AWE) [8] is a method for enhancing the sampling accuracy of the molecular dynamics simulations of protein systems. It partitions the conformational space of a protein into cells and creates a fixed number of simulation tasks or "walkers" in each cell. Every walker is assigned a probabilistic weight such that they provide an unbiased sampling of the conformational space. The sampling efficiency is further improved by utilizing a large number of short simulation steps. The logical structure of the workload in AWE is outlined in Figure 4.10.

Construction and Operation. The work in [8] describes the implementation of AWE using the Work Queue framework and demonstrated its scalability in harnessing 3500 cores from heterogeneous resources in multiple distributed computing platforms [8]. AWE was run on the WW protein domain using 20 walkers, resulting in 12,000 tasks per iteration. There was 40.5MB of common data and each task's unique input files totaled 75KB.

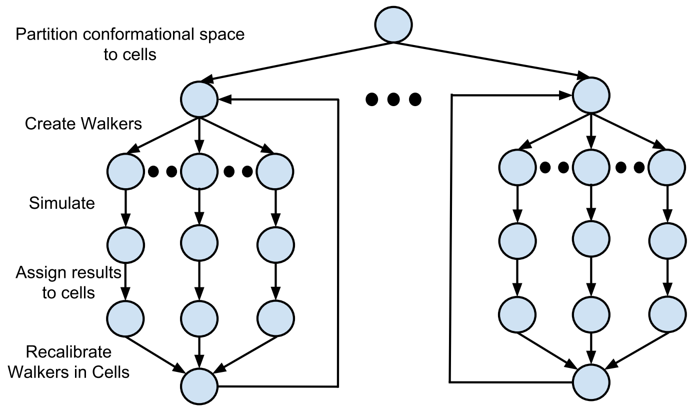

Partition conformational space to cells
Create Walkers
Simulate
Assign results to cells
Recalibrate Walkers in Cells

Figure 4.10. Logical structure of AWE.

The AWE implementation involved the construction of a Work Queue master program that contained the sampling algorithm and logic for (1) preparing walkers, (2) assigning walkers to cells, (3) dispatching walkers to Work Queue workers for parallel execution, (4) extracting statistics from the outputs of completed walkers, (5) determining if additional sampling is required, and (6) determining if walkers need to split or merged for the next iteration of sampling. Figure 4.11 outlines the workings of the implementation of AWE using Work Queue.

Observations. The AWE framework was deployed on heterogeneous resources aggregated from a variety of platform to achieve the scale needed to produce the desired throughput. The platforms used were the HPC clusters maintained at Notre Dame and Stanford, the Condor pools at Notre Dame, Purdue, and University of Wisconsin-Madison, the Microsoft Azure cloud platform, and the Amazon AWS EC2 platform.

The resources allocated for the operation of AWE in [8] were heterogeneous in their availability (on-demand vs queued), processing hardware (CPU vs GPU), processor architecture (x86 vs x86_64), and operating systems (Windows vs Linux). The impact of the heterogeneity on the operation of AWE is observed in Figure 4.12 that plots

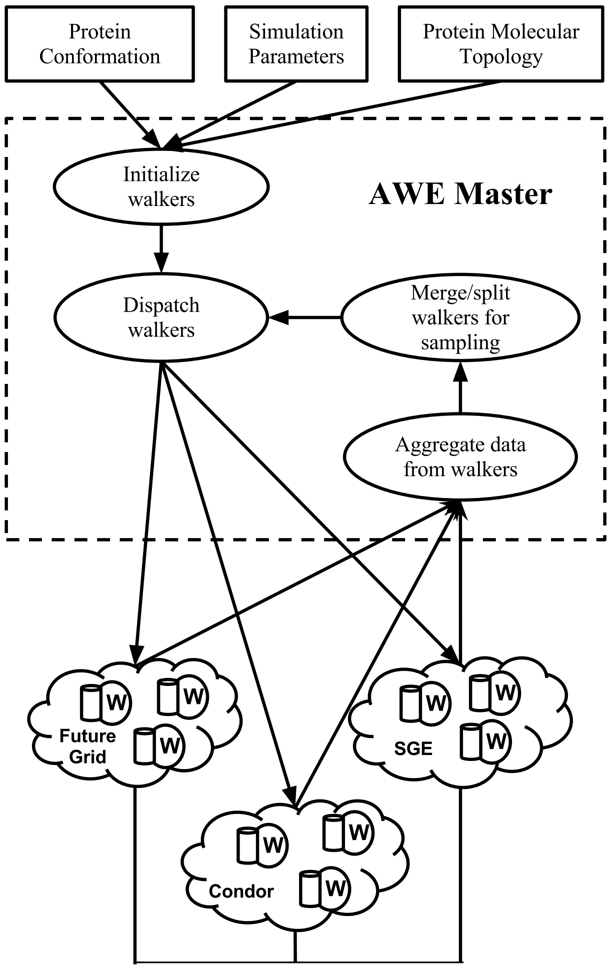


Figure 4.11. Overview of the AWE algorithm and its implementation using Work Queue

the distribution of the task execution times on the deployed platforms. In addition, AWE handed the variation in resource availability by harnessing resources as they become available and being fault tolerant to resources being terminated.

AWE handled operation on heterogeneous hardware, processor architecture and operating system by explicitly specifying its dependencies (libraries, scripts, and programs) and providing compatible versions of the dependencies for each of the operating environments of the deployed resources. This required support from the underlying middleware, Work Queue, to implement the transfer of the version of the

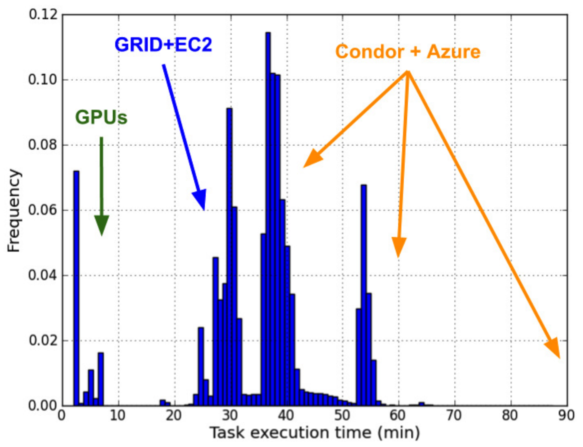


Figure 4.12. Distribution of the execution time of AWE as presented in [8].

dependency based on the operating environment reported by the workers running on the resources.

At the same time, the explicit specification of all the dependencies enabled Work Queue to implement and manage the caching of the dependencies at the workers. This caching strategy enabled AWE to minimize the data transfer costs from the AWE master to the workers which involves the transfer of 34 MB of program files (task initialization) and 100 KB of input files (task allocation) for each task. Specifically, the caching of data at each Work Queue worker lowers the large cost of transferring the program files for task initialization only when a new worker must be initialized. In our operation, we observed that less than 2% of all the tasks sent required task initialization, while the remaining 98% were task allocations. In addition, the overall transfer overhead was found to be small due to caching: the total transfer time (9.3 hours) was about 2.1% of the overall runtime (433 hours).

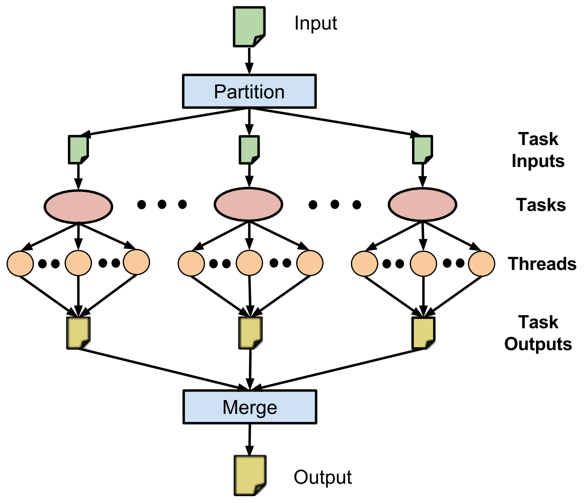


Figure 4.13. Logical structure of Work Queue implementation of the workload operated using Bowtie.

4.7 Elastic Bowtie

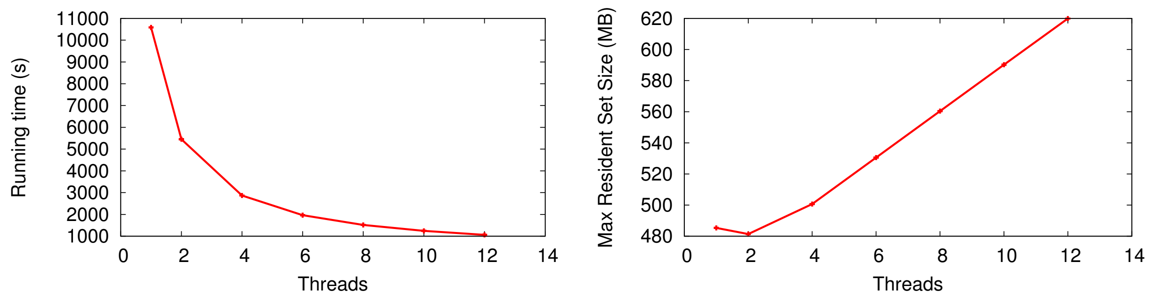
The alignment of sequenced reads of genomes is an important process in the field of genomics. This is because the sequence alignment enables genomes to be compared and contrasted in their evolved and current evolved functions and structure. Alignment refers to the process of arranging the sequenced reads of the genomes such that this comparison can be correctly performed. The alignment of a query sequence is typically performed by comparing against a reference genome such as the human genome.

Function. A number of computer science techniques have been applied in providing optimal solutions to performing alignment of a given query sequence. One of the well-known techniques applies string matching using the Burrows Wheeler Transform (BWT) [28] that originally presented a lossless algorithm for data compression. The BWT based alignment tools are well-suited and optimized with low memory footprint for the alignment of short reads against large reference genomes.

Construction and Operation. Bowtie is a bioinformatics tool that applies

BWT in aligning genomes [69]. Elastic Bowtie is a Work Queue based implementation that partitions and distributes the alignment of sequencing reads to a multiple nodes. Elastic Bowtie is essentially a wrapper script around Bowtie that manages the partition, submission, and aggregation of the tasks that perform alignment. It delegates the alignment of the sequencing reads of the genome to the Bowtie tool which gets invoked as part of the tasks run at the Work Queue workers.

Observations. We ran Elastic Bowtie on a query data that was a subset of the Culex quinquefascaiatus mosquito genome using resources in the HPC cluster maintained at Notre Dame. The query data consisted was about 6GB in size while the reference genome was 400 MB in size.



(a) Running time (b) Memory usage

Figure 4.14: Comparison of the running time and memory footprint of Bowtie with different number of threads when operating on a set of sequences in the Culex quinquefasciatus genome totalling 6GB.

In our operation of Elastic Bowtie, we found the Bowtie tool can leverage thread-level parallelism during execution to improve runtime performance. This implies that Elastic Bowtie exhibited multi-level parallelism where the individual tasks partitioned from the workload can be exhibited in parallel, and in turn each of the individual tasks can be executed using multiple threads running in parallel.

We observe the following in the operation of Elastic Bowtie:

- Similar to other elastic applications in this chapter, as the number of tasks created for operation increases, the operating time first decreases due to the increase in parallelism and then increases due to the startup, partition, and merge overheads. This effect is studied in detail in the next chapter.
- As the number of cores used by the tasks increases, the operating time decreases due to the increase in parallelism but the memory footprint increases from the execution of multiple threads. This effect can be observed in Figures 4.14.
- Given the configuration of the instances (number of cores and RAM), there lies a sweet spot in the decomposition of the workload into multi-core tasks. The presence of the sweet spot is due to the trade-offs between (1) increased thread-level parallelism and thread-level overheads such as increased memory consumption, and (2) increased task-level parallelism and task-level overheads such as the overheads in the partitioning and merging of tasks.
- As the number of instances allocated for operating the multi-core tasks increases, the operating time first decreases due to the increase in parallelism and then increases due to the data transfer overheads. This effect is studied in detail in the next chapter.
- There lies a sweet spot in the operation of the workload as multi-core tasks on multiple compute instances due to the trade-offs between the increase in task-level parallelism and increase in the overheads of data transfer and task initialization.

4.8 Lessons Learned

From our experiences in building the elastic applications described in this chapter, we derive general guidelines for the design and development of elastic applications. While these guidelines are not exhaustive, we believe they are a necessary and useful first step in helping developers build efficient elastic applications.

Abolish shared writes. The use of a shared file to write and aggregate the outputs of tasks during execution are prone to locking overheads and failures as shown in Section 4.4. Elastic applications must therefore implement dedicated and distributed write accesses where files are created and written locally at the site of the task execution. These files can then be transferred from the execution sites (allocated resources) and aggregated at the controller (master). This also allows the storage capabilities at the allocated resources to be utilized effectively. Further, distributed write accesses isolate the performance characteristics and failures of the individual resources thereby minimizing their impact on the overall performance of the application. We showed the benefits of dedicated and distributed write accesses in lowering failures in E-MAKER in Section 4.4. In addition, such accesses simplify the execution environment by avoiding the requirement of a shared file system, and hence improve the ease of deployment.

Keep your software close and your dependencies closer. Elastic application are often deployed on resources with diverse operating environments. To effectively utilize these allocated resources irrespective of their operating environments, elastic applications must transfer and setup the execution environment of each task on the allocated resources. That is, the software components and dependencies of each task, such as executables and libraries, must be encapsulated in the task inputs transfered to its execution site. We applied this technique in the elastic applications to successfully harness resources without imposing any assumptions or requirements on their operating environments. Further, this results in high ease of deployment since the user can deploy and run the application on resources or environments of his choice without any additional effort.

Synchronize two, you make company; synchronize three, you make a crowd. Elastic applications with dependencies between iterations or sets of tasks require synchronization mechanisms to maintain these dependencies. One such mechanism is the global synchronization barrier that span the entire set of concurrent tasks in the application. However, such global barriers introduce inefficiencies in the presence of heterogeneous resources with diverse performance characteristics and adversely impact the time to completion. Therefore, elastic applications must diligently isolate the synchronization requirements to the smallest feasible set of tasks. The use of this technique and its effects in lowering the time to completion were described in Section 4.3. We also observed from the evaluations in Section 4.3 that removing the global barrier yields other benefits, such as lower transfer overheads.

Make tasks of a feather flock together. Ensemble workflows, where a set of independent simulations or computations are run and aggregated as part of a scientific study, can be implemented as elastic applications. In such instances, the outputs of each task or a cluster of tasks contribute directly to a scientific result or output. Elastic applications of ensemble workflows must therefore cluster and prioritize the execution of sub-workflows or tasks that immediately contribute to the scientific output expected during their execution. We showed the application of this technique in the FAW framework (in Section 4.5) to enhance its scientific throughput. Another benefit of improving the scientific throughput through such techniques is that it allows useful scientific output to be obtained quickly even with resource allocations of smaller sizes or shorter durations.

Seek simplicity, and gain power. The choice of the programming abstraction for elastic applications plays a significant role in achieving scale and good performance. In our construction of the elastic applications presented in this chapter, we employed Work Queue that offers a simple and essential set of interfaces to implement and run programs in a master-worker framework. The simplistic and minimalist design of Work Queue requires applications to explicitly (i) decompose workflows into tasks, (ii) specify the inputs to be transferred to the workers for each task, and (iii) aggregate the outputs of completed tasks. However, these explicit requirements allowed applications to harness heterogeneous operating environments, manage and cache data across the allocated resources, and isolate failures. In other words, the sophistication in fault-tolerance, elasticity, handling heterogeneity, and data management directly follows the use of a simple and minimalist interface.

TABLE 4.2

SUMMARY OF THE ESTABLISHED GUIDELINES AND THEIR EFFECTS ON PERFORMANCE

Guideline	**Improves**
Abolish shared writes	Time to completion Cost of failure Ease of deployment
Keep your dependencies closer	Ease of deployment
Synchronize two, you make company; synchronize three, you make a crowd	Time to completion Transfer Overheads
Make tasks of a feather flock together	Scientific throughput
Seek simplicity, and gain sophistication	Transfer overheads Cost of failure Ease of deployment
Model before scaling new heights	Time to completion Ease of deployment

Build a model before scaling new heights. Elastic applications run large computations by decomposing them into tasks. The decomposition of tasks allows concurrent execution but incurs transfer overheads. This decomposition also dictates the size of resources that achieve optimal running time. Therefore, it becomes imperative to formulate a model that captures the effects of task decomposition on the run-time performance of the application as shown in Section 4.2. This model must be incorporated in the application and used to (i) drive the decomposition of the workflow into tasks, (ii) inform the user of the estimated performance for a given input, and (iii) guide the user in allocating resources for execution with a given input. Such models are especially useful when the applications are designed to achieve scale and are deployed on resources that incur monetary costs to the user.

Table 4.2 summarizes the presented guidelines and their effects in improving the operation and performance of elastic applications.

4.9 Conclusion

We studied a cross-section of scientific applications constructed and operated as elastic applications. Across all the studied applications, we noticed the primary challenge to their efficient operation was in the decomposition of the workload into tasks. The decomposition determines the concurrency in the task executions, the costs to partition and aggregate tasks at the master, and the transfer overheads associated with the tasks. These runtime components, in turn, dictate the overall performance of the application. In other words, the manner of task decomposition directly impacts the runtime performance of the elastic applications.

The second challenge observed across the studied applications was in the allocation of the right size of resources. Currently, there does not exist a clear mechanism to describe the resource requirements of the application. This affects the users of these applications who are left to predict or assume the resource requirements of the application for the given workload and deploy an appropriate size of resources. This in turn results in either overprovisioning of resources leading to enormous cost overruns or underprovisioning resulting in poor runtime performance. To overcome these inefficiencies in their deployment and operation, elastic applications must provide guidance on their resource requirements and runtime performance.

These challenges in the design and operation of elastic applications can be summarized as the following:

- What is the decomposition strategy in terms of the number of tasks that achieves time- and cost-efficient operation?
- What is the scale of instances that must be provisioned for achieving time- and cost-efficient operation?

CHAPTER 5

MIDDLEWARE TECHNIQUES

5.1 Introduction

In this chapter, I explore techniques in the middleware to improve the performance and cost-efficiency of elastic applications and support the application-level techniques in achieving these goals. In doing so, I argue against common wisdom that has required middleware to be designed such that they completely hide the details about the underlying execution environment from the applications. I demonstrate how applications can actually benefit from selective information about the execution environment being exposed by the middleware. At the end of this chapter, I also show how techniques that improve the data transfer and management semantics in the middleware help improve the overall cost-efficiency and performance of elastic applications.

Middleware are abstractions that simplify the design and operation of applications by hiding the complexities in the underlying execution environment. They are considered an integral part of most software stacks primarily due to their role in enabling application developers and operators to build, deploy, and operate applications without concern for the details and complexities of the execution environments, such as scheduling, failures, and heterogeneity of resources.

The origins of middleware can be traced to the earliest batch computing systems, such as mainframe computers, that used abstractions to manage centralized pools of resources shared across multiple users and processes. Middleware became a necessary component in multi-core and -threading environments for handling the low-level I/O and resource management. This trend continued with the advent of parallel and high performance computing that resulted in the birth of middleware that managed a network of several machines. The middleware provided communication primitives for coordinating computations across the networked machines. An example is MPI that provided interfaces and mechanisms to manage the execution of a group of communicating processes over distributed resources.

Over the last two decades, the role and functions of middleware have expanded in distributed computing systems to include the transparent handling of the unpredictable failures, latencies, scheduling, placement of data, and locality. Figure 5.1 describes the outline of the functionalities offered by middleware for the operation of elastic applications in distributed computing environments. The middleware in distributed computing systems can appear in many forms such as batch submission and scheduling systems (e.g., Condor [111], Maui [63]), distributed execution frameworks (MapReduce [40], Dryad [62]), and data processing and storage abstractions(Spark [129], DynamoDB [41]).

5.2 Current approaches and drawbacks

Middleware in parallel and distributed computing systems [53, 62] offer facilities that enable developers to delegate the runtime decisions, such as the concurrent execution of the workload, to the middleware. The use of such facilities has proved beneficial in environments such as batch processing systems and supercomputers, where the pool of allocated resources are fixed, shared across multiple processes and users, tightly controlled and monitored, and entirely administered by the middleware. In such shared operating environments, the goals of operation are to provide availability, reliability, and fairness in resource allocations for competing processes and users. The middleware help achieve these goals by scheduling, managing, and

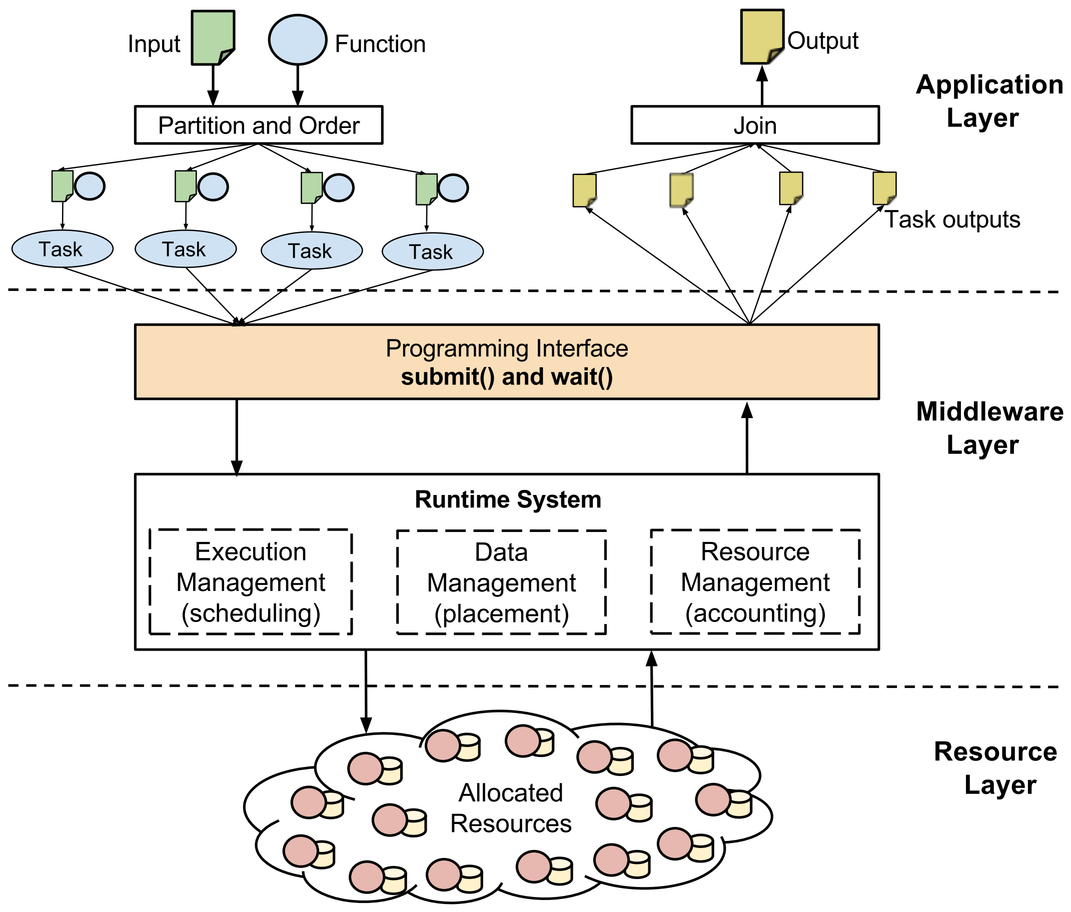


Figure 5.1. Outline of the architecture and operation of applications using middleware in distributed computing environments.

directing the operation of the processes submitted for execution.

The goals and characteristics of these environments also resulted in the widespread adoption of the following principles in the design of middleware: (1) information about the underlying resources were completely hidden from applications since the middleware were tuned to manage the operation of applications on the resources they administered in a manner that was optimal to the operators of the resources, and (2) the data transfer costs were considered inconsequential and ignored since the resources for operation were provisioned at a single site and often networked using high-bandwidth communication links.

However, the above principles of design for middleware translate poorly to environments characterized by exclusive, dedicated, and metered deployments for every execution. These environments are common in the current generation of distributed computing where the operators and end-users can deploy the applications on any currently accessible resources from a variety of distributed computing platforms. This implies that middleware cannot be tuned to a particular class or type of resources for operating the applications. Further, the cost of data transfer becomes significant since resources can be deployed across geographically distributed networks connected by low-bandwidth and metered communication links.

Consider Hadoop [53], a widely adopted middleware for executing concurrent and data-intensive workloads expressed using the MapReduce paradigm [38]. Hadoop relies on a distributed file system, such as Hadoop Distributed File System (HDFS) [103], for managing data during operation. HDFS partitions the input data and stores the partitions across the nodes provisioned for operation. HDFS arbitrarily partitions the data into blocks (default size of 128 MB) regardless of the workload and the concurrency feasible during its operation. HDFS leaves the optimal tuning of the block sizes to the operators (and not the users) of the cluster based on the characteristics of the resources in the cluster and the expected characteristics of the workloads that will be executed. It does not however provide guidance on an optimal partitioning strategy for executing the defined workload.

Hadoop is also designed with the principle that the movement of computations to data is cheaper and faster. However, it does not optimize the transfer and movement of data. Instead, it assumes the resources are provisioned within a single network boundary and that data transfers are cheap, fast, and without failures. As a result, it recommends and relies on copying and moving every input data to multiple resources to achieve replication and guard against loss of data.

In summary, the partitioning strategy in Hadoop is fixed, globally enforced on every application, and based on the assumption that data migration can happen without incurring costs and overheads. The use of arbitrary and global policies for operation results in cost-inefficiencies in environments where each instance of the application is deployed on resources exclusively dedicated for their operation. Instead, the workloads must be partitioned and the resource requirements must be estimated according to the characteristics of the deployed operating environment. These decisions must be made in every deployment of the application since the operating environment can vary between deployments. Further, these decisions must be revised and adapted during operation since the characteristics of the operating environment, such as network bandwidth, are liable to dynamically change.

5.2.1 Solutions

In this chapter, I argue that middleware must expose selective key information about their operating environment to the applications. That is, middleware should gather and expose information on key resources, such as the number of cores, the size of memory, and the network bandwidth in the operating environment. By exposing such information, I show that middleware can enable applications to make intelligent decisions on their operating parameters that will achieve efficient operation in the currently deployed environment.

I then evaluate a technique focused on minimizing the costs and overheads associated with data transfer. The technique uses a hierarchical configuration and minimizes the transfer costs when data movement spans across network boundaries and involves resource failures when data is in transit and at rest. I evaluate the benefits of this technique using an experimental run of the AWE elastic application at scale on over 4500 cores provisioned from multiple geographically distributed platforms containing commodity hardware.

5.3 Expose selective information about resources

In the previous chapter, I argued and demonstrated that runtime performance and cost-efficiency are maximized when applications directly exert and regulate control over the partitioning of their workload. This is because the applications explicitly know the characteristics of the workload, such as data dependencies, transfer requirements, and partitioning overheads. In addition, this approach provides flexibility to the users in deploying and operating the applications on resources for their choice without being tied to a particular platform or operating environment.

The performance and operating costs of applications are primarily determined by the size of the partitions of the workload and the cost of distributing these partitions for concurrent execution. I show that the decisions on these parameters benefit from knowledge of the resource capacities, particularly CPU, memory, and network bandwidth. First, the number of available CPU cores dictates the physical concurrency achievable in executing the workload. Second, the size of memory at the available instances determines the thresholds that incur minimal I/O overheads during execution. Finally, the network bandwidth determines the costs incurred by the data transfers required to setup concurrent execution. Therefore, knowledge of the available CPU, memory size, and network bandwidth enable applications to determine the overheads and costs of partitioning and distributing their workload using a certain configuration.

To enable the applications to partition appropriately for the deployed environment, we propose that middleware must provide interfaces through which the information about the resource capacities are exposed to the applications. This approach preserves the benefits and portability advantages of using middleware to manage the complexities of distributed execution environments. We note that the information exposed through the interfaces in the middleware must be limited specifically to resource capacities. Other significant details and information about the execution environment such as the heterogeneity, failure, and performance characteristics of resources must remain hidden by the middleware.

5.3.1 Evaluation

To study the benefits of middleware exposing selective information about the cores, memory, and network bandwidth, I consider their impact on the performance of E-Sort and E-MAKER.

We begin by considering the impact of the size of memory on the performance of E-Sort. The performance of E-Sort is impacted by the size of the memory since it determines the amount of data that can be processed at a single instance without incurring I/O overhead related to hard disk usage. On the other hand, the network bandwidth does not impact the performance of E-Sort with different partition sizes since the amount of data to transfer remains the same for any partition size.

Figure 5.2 compare the estimated and actual running time of E-Sort on an instance with 12GB memory. As the size of the data sorted in the instance increases beyond 12GB, the overheads from swapping to the hard disk increases resulting in an increase in the overall running time. This effect can be observed in the actual running time trending above the estimated values.

In order to avoid the effects of I/O overheads on the runtime performance of E-Sort, the partitioned tasks must only sort data that is smaller than the size of the memory available at the execution sites. To achieve this behavior, the middleware must first expose information about the size of the available memory at the instances provisioned for operation. E-Sort can apply this information as a bound on the size of the partitions to use in operation. Note that the size of the available memory only serves as a bound and that the partitions for operation must be determined based on the measured overheads of partitioning of inputs, execution of the partitioned tasks, and the merging of the outputs. Figure 5.3 shows the lower bounds on the partitions

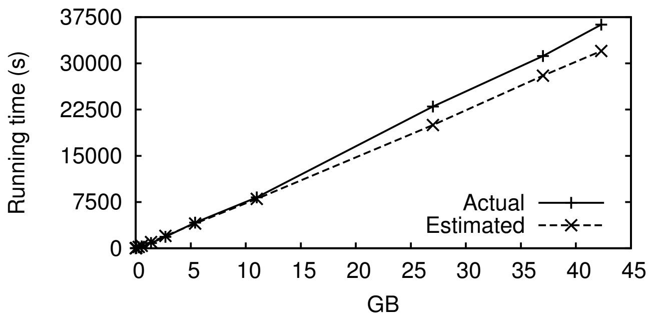


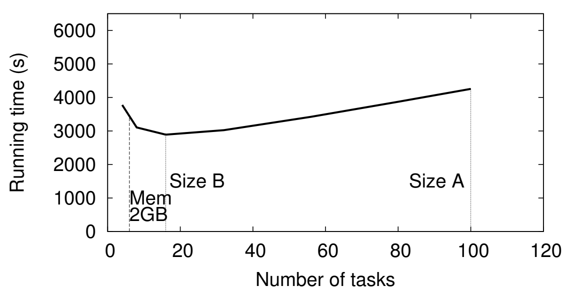
Figure 5.2. The estimated and actual running time of GNU Sort for various input sizes on an instance with 12GB RAM running in the Notre Dame SGE platform.

Note the actual running time incurs I/O overheads when the size of the data processed at the instance exceeds 12GB. This effect can be observed in the actual running time deviating away from the estimated values at 12GB.

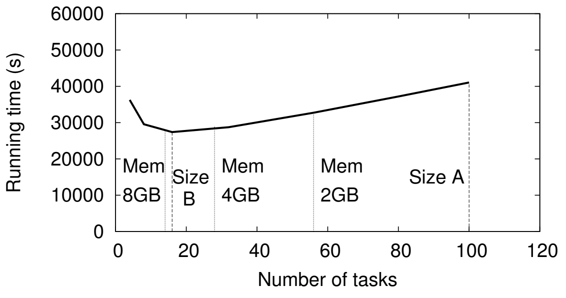
determined based on the size of the memory at the provisioned instances and the partitions estimated to achieve optimal running time for two different input data.

On the other hand, E-MAKER is impacted by the resource capacities differently than E-Sort. In E-MAKER, the query data set of genomes is partitioned for concurrent operation while the reference data set is fixed and transferred without being partitioned to each instance where the partitioned query set is annotated. This was done to avoid the complexity associated with merging the outputs of tasks when both the query set and reference set are partitioned. As a result, the reference set - being larger than the query set - dominates the memory consumption. Therefore, the size of memory does not impact the runtime performance of E-MAKER in a discernible manner like in E-Sort.

The network bandwidth affects the performance of E-MAKER in the transfer of the reference data set and the software dependencies to every instance used for operating the concurrent partitions. The network bandwidth impacts the running time of E-MAKER since it determines the time spent in the transfer of the data to the instances. For example, when the network bandwidth is high, E-MAKER can



(a) Data size: 11GB



(b) Data size: 111GB

Figure 5.3: Illustration of the choices on the partitions for sorting records in a file of size 1 GB.
In this figure, Size A corresponds to the partitions chosen when only information about the number of cores is available. Size B corresponds to the partitions when information about the task execution times and the merge overheads are available. Here Size B represents the optimal number of partitions for the considered execution environment.

utilize a higher the number of partitions and instances since the overheads of data transfer are minimal. Therefore, E-MAKER must have access to information about the network bandwidth in its deployed environment to determine the partitions for operation. Figure 5.4 illustrates the choices made based on whether information about the network bandwidth is available. Figure 5.5 shows the impact of different network bandwidth values on runtime performance. This figure emphasizes the need for measuring the network bandwidth and providing this information to E-MAKER so it can determine the partitions that achieve efficient operation.

5.4 Hierarchical Data Distribution

The previous chapter described the implementation of AWE using the Work Queue framework and its scalability in harnessing 3500 cores from heterogeneous resources in multiple distributed computing platforms [8]. While we found Work Queue to provide the support and controls for operating elastic applications, we also noticed a few limitations in its flat master-worker architecture. First, the master

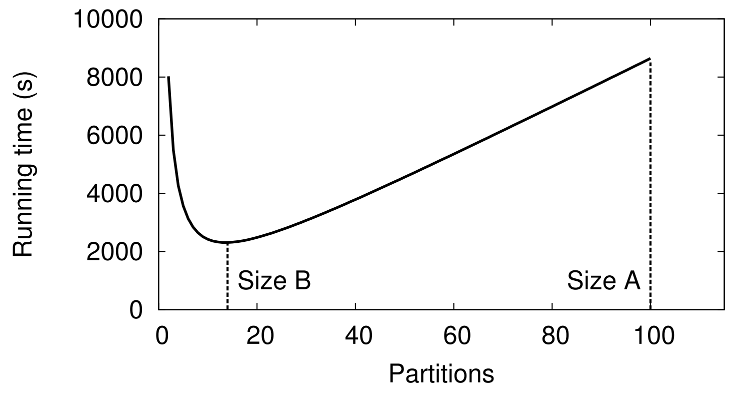


Figure 5.4. Illustration of the choices on the partitions for annotating 200 contigs of the Anopheles gambiae genome.

In this figure, Size A corresponds to the partitions chosen when only information about the number of cores is available. Size B corresponds to the partitions when information about the network bandwidth and the task execution times are available. Here Size B represents the optimal number of partitions for the considered execution environment.

must transfer the input and output data of the tasks to and from its workers. As a result, scalability is constrained by the network bandwidth available at the master. Further, if the data transfer between the master and the workers happen over metered communication channels (e.g., data transfer in commercial cloud platforms such as Amazon AWS is metered), it adds to the monetary costs incurred by the operators of the applications. Second, the master must handle the failures and terminations of the worker and spend bandwidth transferring data and re-establishing the execution environment at workers with transient failures. Finally, the number of workers that can connect to a master can get dictated by administrator policies on resource usage, such as the number of TCP connections that can be maintained by any one individual process running in a system.

To address these limitations of the traditional flat master-worker architecture, Work Queue was redesigned to provide a hierarchical configuration where a single master can be served by multiple levels of sub-masters with the workers connected to

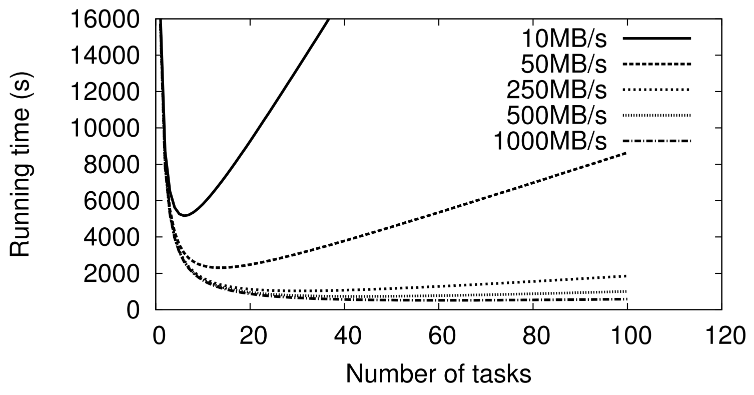


Figure 5.5. Estimated running times for annotating 200 contigs of Anopheles gambiae genome shown for different network bandwidth values.

the sub-masters at the lowest level. The sub-masters are referred to as *foreman* in the Work Queue hierarchy. Figure 5.6 presents an outline of the hierarchical configuration using multiple foreman in Work Queue.

The foremen in Work Queue receive files from the master and cache them at their execution site. The workers deployed on the provisioned resources are programmed to connect to a foreman that is usually deployed in the same platform as the workers. The workers treat the foreman as their master and connect to the foreman to receive tasks to run. The foreman dispatches tasks by transferring the input files received from the master and cached at its execution site. The foreman reduces data transfer overheads by leveraging the well-provisioned intra-platform bandwidth and eliminating the repeated transfer of common data between the master and the workers. It also eliminates these overheads in environments with high worker volatility (e.g., workers in Condor often get shut down and migrated elsewhere when a higher priority user starts using the resources) by getting rid of the traffic between the master and the workers that reconnect after failure. A detailed description and analysis of the Work Queue hierarchy can be found in [13].

The key takeaway from the use of the hierarchical Work Queue architecture is that

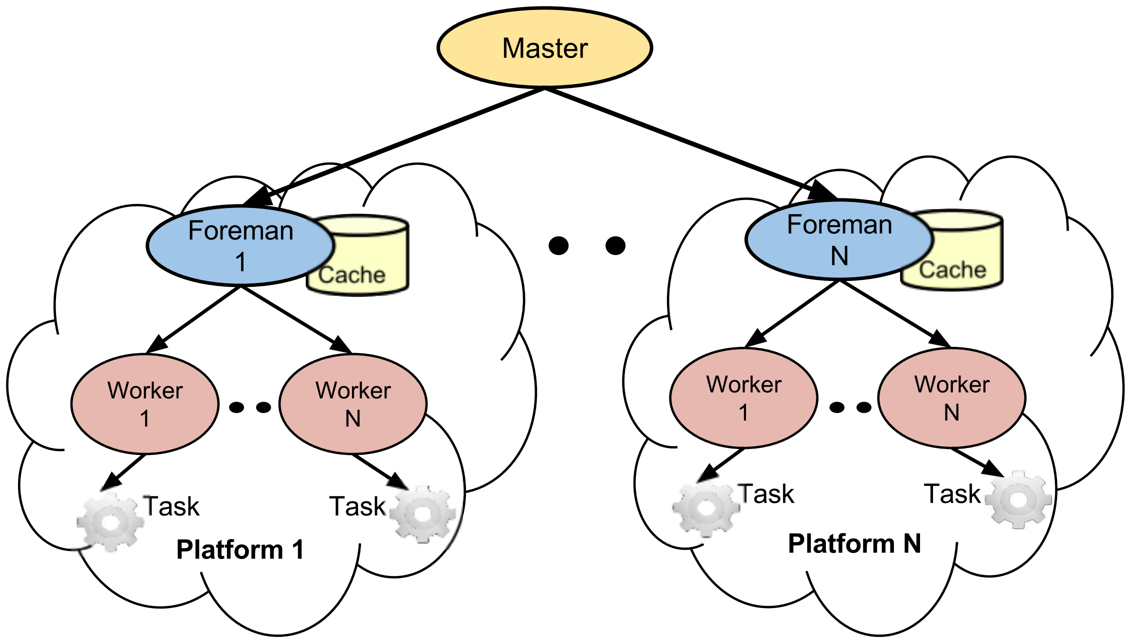


Figure 5.6. Outline of the hierarchical configuration in Work Queue.

middleware must consider the costs and overheads of data transfer and implement mechanisms to minimize them. In Work Queue, we show that the use of a hierarchical configuration of masters achieves this goal in minimizing the cost and overheads of data transfer during operation of the application.

We proceed to show the hierarchical Work Queue in action and evaluate the improvements it enables in the operation of the applications.

5.4.1 Case study using AWE

We use hierarchical Work Queue to improve the scale and fault-tolerance in the operation of AWE. In this study, we ran AWE on the WW protein domain using 20 walkers, resulting in 12,000 tasks per iteration. There was 40.5MB of common data and each task's unique input files totaled 75KB. Table 5.1 describes the input data and parameters used in our experimental study.

TABLE 5.1

PARAMETERS USED FOR THE EXPERIMENTAL AWE RUN

AWE Parameters	**WW domain**
Iterations	3
Walkers	20
States	601
Common input size per task	40.5MB
Unique input size per task	75KB
Output size per task	82KB
Tasks per iteration	~12000
Task Execution Time	~30 minutes

We used resources from four clusters - Notre Dame's High Performance Cluster, the ND Condor pool, FutureGrid's Sierra cluster (at San Diego Supercomputer Center), and FutureGrid's India cluster (at Indiana University, Bloomington). At each independent cluster (ND-HPC, Sierra, and India), we ran a single foreman on the head node and set up single-slot worker processes for every core in our allocation: 200 workers at ND-HPC and 800 workers across the two FutureGrid sites.

We submitted 5000 workers to the ND Condor pool of which approximately 3400 were seen over the course of the experiment. Due to a department policy there is a limit of 1024 simultaneous TCP connections on each Condor node, so we allocated one foremen for every thousand Condor workers. Each foreman ran on its own machine at the same data center as the majority of the Condor pool.

Scale. We ran the experiment for about 15 hours, completing three iterations. Figure 5.7 shows the concurrency we achieved during the experiment. The valleys correspond to synchronization barriers at the end of each iteration, while at our

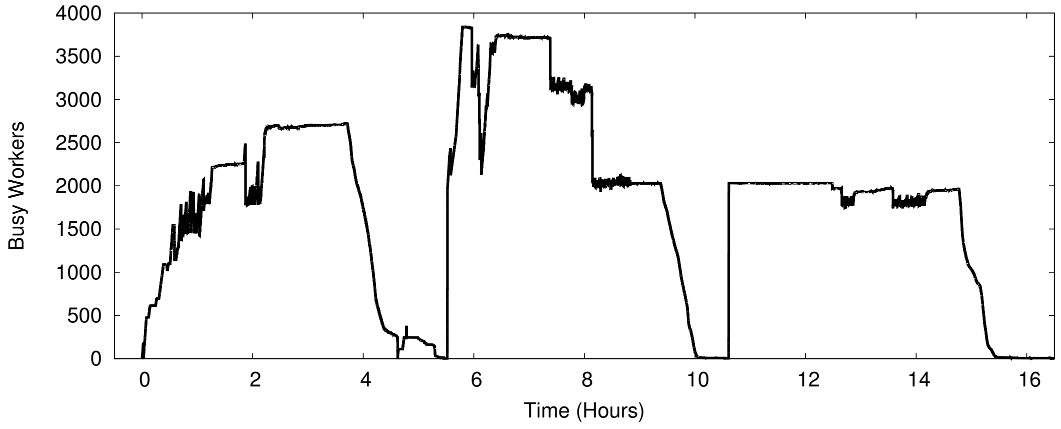


Figure 5.7. Busy workers over the duration of the AWE run.

peak (around hour 6) we saw 3862 workers simultaneously executing AWE tasks. The drop in concurrent workers at hour 8 occurred because of the termination of allocated resources in FutureGrid (due to limits on resource usage) and Condor (due to contention for resources). Table 5.2 shows our measurements of this experiment, including the number of tasks dispatched by the master and each foreman, the total failures of foreman and its workers, the average bandwidth observed between each component, and the cumulative data sent by each entity.

Failures. Over the course of a 15-hour experiment using thousands of resources spanning multiple locations some number of failures are inevitable. Causes may include network disruptions, local disk failures, and scheduling policies. Examining the results in Table 5.2 we noticed the number of tasks dispatched by the master to a foreman was often higher than those dispatched by that foreman: this indicates that the foreman failed before being able to dispatch all the tasks assigned to it. On closer inspection we found that the foreman failed multiple times in a very short timeframe, indicating a short-term resource failure rather than transient network fluctuations. Worker failures had a much more varied set of causes, including resource failures, network disruptions, and terminations due to cluster scheduling policies.

Bandwidth and Data Usage. Table 5.2 shows the average bandwidth measured between the master and foremen, as well as between each foreman and its workers. The master to foreman bandwidth is consistently smaller than the foreman to worker bandwidth. The difference was especially pronounced in the case of our foreman running at the Sierra site in San Diego, over 1800 miles away.

Table 5.2 also shows the data sent by the master and foremen, as well as an estimate of the data a master in a flat configuration would have sent when running the same experiment. This estimate was derived by adding the data sent by each foreman and removing the data resent due to foreman failures, since those retransmissions do not exist in a flat configuration. In comparison to the hypothetical flat master configuration, we see on average a 96% reduction in data transmitted by the hierarchical master.

TABLE 5.2

STATISTICS FOR THE AWE RUN ON WW DOMAIN USING A MASTER-FOREMAN-WORKER HIERARCHY

	Max Workers	**Tasks Dispatched**		**Resource Failures**		**Average BW (Mbps)**		**Total Data Sent (GB)**			
		M	F	F	W	M -> F	F -> W	M -> F	F -> W	Est Flat M	Savings (%)
ND-Condor											
Fmn 1	447	9279	9064	7	59	178	809	1.0	74.6	33.3	97.8
Fmn 2	915	14614	14033	13	294	135	900	1.7	168.9	74.6	97.7
Fmn 3	499	2451	1657	2	9	124	2163	0.3	53.5	21.4	98.6
Fmn 4	688	3146	2431	2	69	137	1481	0.4	69.0	28.3	98.6
Fmn 5	805	18715	18428	11	24	124	891	1.9	191.7	50.2	96.2
ND-HPC											
Fmn 1	200	7902	6855	50	6	139	492	2.6	179.4	22.5	88.4
FutureGrid											
Fmn Sierra	678	6793	4905	49	1120	11	449	2.6	186.4	107.6	97.5
Fmn India	88	1747	1747	3	75	267	392	0.3	16.1	12.3	97.6
Total	3862	64647	59120	141	1656	-	-	10.8	939.6	350.2	96.9

5.5 Conclusion

In this chapter, we argued against the time-tested principle that advocates middleware must hide all complexities and details about the underlying execution environment from the applications. We argued this by showing how selective information exposed by the middleware to the applications enables efficient operation. The experimental illustrations also highlight and support the key idea presented in this work that applications must directly determine the parameters that impact their runtime characteristics and behavior. These parameters must be determined using the information exported by the middleware about the underlying execution environment. It might also benefit to clearly demarcate the management boundaries of the middleware when building elastic applications. Middleware must be involved only in the management of the resources and the execution environments (such as handling failures and maximizing the utilization of resource capacities) while leaving the management of the operation of the workloads to the respective applications.

We also showed that middleware cannot neglect the costs and overheads of data movement and must incorporate techniques to minimize them. We experimentally evaluated a hierarchical data distribution technique in the middleware that minimized the costs of data transfers and showed how this translated to better runtime performance and efficiency of the applications.

In the next chapter, we explore application-level techniques that utilize the information exposed by the middleware about the underlying operating environment and determine the operating parameters for running their workloads in a time- and cost-efficient manner.

CHAPTER 6

APPLICATION-LEVEL TECHNIQUES

6.1 Introduction

In this chapter, we study and build techniques in the application layer to address the challenges to the efficient operation of elastic applications. We consider efficiency in terms of the runtime performance and the monetary costs incurred during operation. In the past, runtime performance was often considered as an important metric for evaluating the efficiency of large applications. However, the consideration of the monetary costs of applications has become a necessity only since the emergence of cloud computing platforms as a viable operating environment for large scale applications. Cloud platforms charge for the usage of resources using a metered "pay-as-you-go" model often rounded to hourly boundaries. Therefore, deployments on cloud platforms demand cost-efficient operation.

As we described earlier, we consider the deployment and operation of applications using resources exclusively dedicated to each instance of the applications. The applications can be deployed and operated on any distributed computing platform accessible to the operators. As a result, the characteristics of the target operating environment (such as execution speed and network bandwidth) are unknown and unpredictable prior to deployment.

In the past, application designers would tune a concurrent application for specific hardware and operating environment (such as supercomputers and high performance clusters) and require the application to be operated in those environments. But if the target hardware is unknown or variable when the application is designed, the application must be self-tuning at runtime. In other words, **elastic applications in distributed computing environments must be self-tuning to be time- and cost-efficient**.

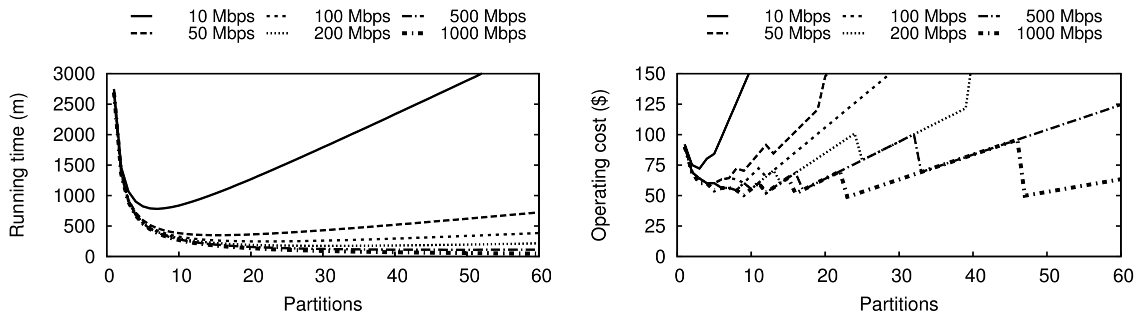
6.2 Analysis of the Challenges to Efficient Operation

The time and cost of operation of elastic applications are determined by the gains achieved in the concurrent execution of tasks and the overheads incurred in the split, merge, and data transfer phases. The gains and overheads are determined by the number of partitions and resources chosen for operation. They are also influenced by the characteristics of the operating environment. For instance, the network bandwidth influences the transfer overheads while the size of the RAM at the provisioned instances influences the overheads of executing the map functions on the partitions.

In this chapter, we focus on operation using the on-demand instances in distributed computing platforms. These instances can be provisioned and terminated at convenience and are metered to incur charges only for the duration of use. To simplify exposition, we assume the instances cost \$1 per hour and incur \$0.01 for every gigabyte of data transferred to and from the instances[2]. Like many metered platforms, we compute the operating costs by rounding the operating times to the nearest hour.

Figure 6.1 illustrates the impact of the partitions and the characteristics of the operating environment on the time and costs of operating E-MAKER on the Anopheles gambiae genome. The plots assume the number of partitions and the instances provisioned for operation are equivalent. From Figure 6.1, we observe the running time and operating costs exhibit varying trends for different network bandwidth. Further,

[2]Our observations remain the same with the prices in commercial platforms such as Amazon EC2.



(a) Running time (b) Operating costs

Figure 6.1: Estimated running time (in minutes) and operating costs (in $) of E-MAKER for annotating the Anopheles gambiae genome.
The estimations are shown for different network bandwidth in this figure.

Figure 6.1b shows the operating costs for various partitions exhibit irregular patterns due to the effects from the use of the hourly boundaries for calculating costs. The operating costs drop at partitions where the time of operation falls to the next lowest hourly boundary. In summary, Figure 6.1 shows that the choice of the number of partitions and instances to provision are critical to the cost-efficient operation of applications.

Figure 6.2 plots the partitions that achieve the minimal operating time and the lowest operating costs in Figure 6.1. It demonstrates that performance and cost cannot always be optimized simultaneously, and so the partitioning must take into account the differing objectives of each user.

Further, the workloads must be partitioned and the resource requirements must be estimated according to the characteristics of the deployed operating environment. This need is illustrated in Figure 6.2 where the partitions that achieve minimal operating costs vary depending on the operating environment. This effect can be inferred from Figure 6.2 where the partitions that achieve minimal operating costs vary depending on the network bandwidth in the operating environment. To achieve cost-efficiency, the workloads must be partitioned and the resource requirements must be estimated according to the characteristics of the deployed operating environment.

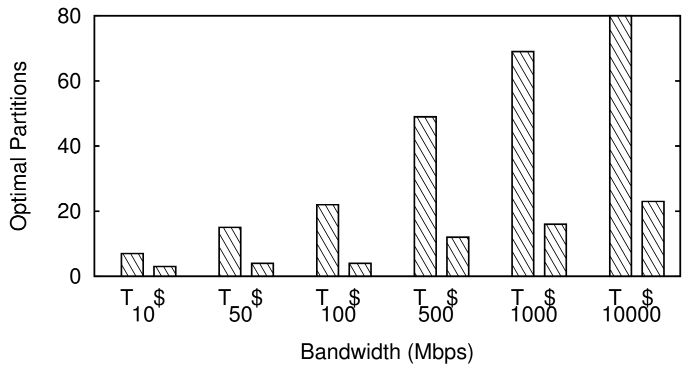


Figure 6.2. Illustration of the optimal partitions that achieve minimum running time and operating costs for annotating the Anopheles gambiae genome under different network bandwidth values.

The running time is denoted by T and the operating costs are denoted by \$.

These decisions must be made in every deployment of the application since the operating environment can vary between deployments. Further, these decisions must be revised and adapted during operation since the characteristics of the operating environment, such as network bandwidth, are liable to dynamically change.

6.2.1 Overview of Application-level Techniques

The performance and costs of elastic applications are determined by the (logical) expression and (physical) realization of concurrency during their operation. The expression of concurrency pertains to the number of partitions of the workload while its realization involves the simultaneous execution of the partitions. In this chapter, we argue that applications must be self-modeling in order to determine and tune their logical and physical concurrency at runtime. The applications must incorporate a model of their operation formulating the gains and overheads of concurrency. We also argue that cost-efficient operation requires applications to explicitly exert control of the partitioning of the workload to tasks, the binding of data to tasks, and the submission of tasks for simultaneous execution.

Using the model and control of the parameters of concurrent operation, the applications tune and adapt their operation according to the characteristics of the deployed environment. The applications first measure the characteristics of the environment that influence their transfer overheads, processing overheads, and I/O overheads, and thereby, their cost-efficiency. The measurements are applied in the model to estimate the overheads of operating the defined workload under the current operating conditions. Using these estimates, the applications determine and adapt their logical and physical concurrency during runtime to achieve cost-efficient operation in the deployed environment.

In summary, we argue and demonstrate the following principles for correctly determining the operating parameters for cost-efficient operation:

1. **Applications must be self-modeling** by formulating and incorporating a model of the performance and overheads of their runtime components.
2. **Applications must be self-tuning** by exerting control over the partitioning and concurrent operation of the workload and dynamically adapting the operation according to the observed operating environment.

6.3 Application-level Modeling

The runtime performance of split-map-merge based elastic applications are determined by the partitioning, task execution, data transfer, and merge components described in Section 3.3. Accordingly, we model the operating time of elastic applications as follows:

$$T_{operation} = T_{partition} + T_{tasks} + T_{data} + T_{merge}. \tag{6.1}$$

Note this model of the running time differs from Amdahl's law [19] in that $T_{partition}$ and/or T_{merge} at the coordinator increases with the number of partitions.

The overheads of the partition $T_{partition}$ depend on the size of the input data N

and the number of partitions K. We model this as a linear relationship:

$$T_{partition} = (a * N) + (b * K), \tag{6.2}$$

where a and b are constants that reflect the costs of reading (input) data and creating a partition respectively.

As we noted in Section 3.3, the merge overheads T_{merge} can vary based on the implementation and characteristics of the workload. Therefore, these overheads are formulated and discussed individually in Section 6.6.1.

The execution time of the tasks T_{tasks} is determined by the size of the input and the partitions. If input N is partitioned into K tasks, the execution time of a task is

$$T_{task} = T(\frac{N}{K}). \tag{6.3}$$

If R is the number of instances provisioned for operation, the execution of K tasks is prolonged by a factor of $\lceil K/R \rceil$ (as only R tasks can be executed simultaneously). Thus, the total execution time of tasks is

$$T_{tasks} = T_{task} * \lceil \frac{K}{R} \rceil. \tag{6.4}$$

The data overheads T_{data} collectively represents the input T_{inputs} and output $T_{outputs}$ transfer overheads. The inputs consists of the software and unique data dependencies of the tasks. The software dependencies include executables, scripts, and libraries are required for execution and are common across tasks. These dependencies can be transfered once and cached for subsequent tasks. In contrast, the unique data dependencies are specific to each task and must be transferred for every execution. These dependencies specify the data for operation in each task.

$$T_{data} = (Data_{in} + Data_{out})/BW, \tag{6.5}$$

$$Data_{in} = size(N) + R * size(software), \tag{6.6}$$

$$Data_{out} = size(N), \tag{6.7}$$

where BW represents the available network bandwidth.

The model in Equation 6.1 is applied to determine the number of partitions K and the number of instances R to provision for operating the defined workload. Further, the estimations from the model enable the applications to tune and adapt the partitions according to the characteristics of the deployed environment, such as network bandwidth, physical memory allocated at the resources, and the processing capacity for operation.

Assumptions: The model in Equation 6.1 assumes a scheduling strategy that operates in the following order: dispatch tasks submitted for execution to the provisioned instances (this includes transfer of input data), wait for tasks to finish execution, retrieve the outputs and results of the executions, and so forth. It is also assumed that all the instances for operation are provisioned at the same time and the workers running on them are connected to the master before the dispatch of tasks begins.

The model for task executions in Equation 6.4 assumes the instances provisioned for operation are homogeneous in their hardware and processing capabilities. In cloud platforms, this assumption is satisfied by provisioning instances of the same size (e.g., in Windows Azure the sizes are small, medium, etc). Further, it assumes that each task consumes a single CPU core.

Finally, the model of the data overheads in Equation 6.5 assumes single-threaded communication where data is transferred to one worker at a time. A multi-threaded mode is helpful when communicating with heterogeneous instances with wide differences in their processing capabilities. Otherwise, we expect the impact from multi-threaded communication on the estimations to be minimal since the bandwidth remains the same while being shared across multiple threads.

6.3.1 Cost-efficiency Metrics

We formulate the operating cost $C_{\$}$ of the applications using the time of operation modeled in Equation 6.1, the instances provisioned for operation R, and the data transfered during operation.

$$C_{\$} = \$_{IH} * R * H_{operation} + \$_{GB} * (Data_{in} + Data_{out}), \tag{6.8}$$

where $H_{operation}$ is $T_{operation}$ rounded to the nearest hour. $\$_{IH}$ represents the cost incurred per instance per hour of use while $\$_{GB}$ represents the cost charged per gigabyte of data transfer to and from the instances. To simplify analysis and provide a general context, we assume $\$_{IH}$ to be \$1 and $\$_{GB}$ to be \$0.01.

We note that favorable trade-offs often exist between the time and cost of operation. For example, in Figure 6.1b, the lowest operating cost (\$60.16) under a bandwidth of 100Mbps is achieved when operating with 4 partitions. However, accommodating a 0.33% increase in the operating cost (\$60.36) for operation with 9 partitions leads to a 50% decrease in the operating time. This is because operation with 9 partitions increases the gains from concurrent executions and lowers the time of operation such that it matches the gains in cost when operating with 4 partitions. The small increase in cost for the operation with 9 partitions results from the increase in the transfer costs for the software dependencies.

In our evaluations, we find it useful to compute and use a metric called *Cost-Time* *product* to consider these trade-offs and assign equal importance to the time and cost of operation.

$$Cost\text{-}Time\, product = C_{\$} * T_{operation}. \tag{6.9}$$

We note this metric is only one of several ways of expressing cost-efficiency since different weights may be assigned to the time and cost of operation based on the preferences of the operators. At the same time, we note the model can be easily extended to provide estimations on the cost-efficiency metrics preferred by operators.

6.4 Application-level Control

The application-level model in Section 6.3 provides estimations on the performance and overheads of the runtime components. However, to regulate the overheads and achieve cost-efficiency, the application must explicitly direct and control the following actions using estimates from the model.

Partitioning workload into tasks: The applications must define the partitioning of the workload into tasks using estimates from the model. The number of tasks created for operation also dictates the overheads associated with the task executions, data transfers, and merge operations. As a result, control over the number of tasks enables applications to lower the running time and minimize the incurred overheads in their deployed environments.

Binding data to tasks: The applications must explicitly bind the data dependencies to the created tasks. This control enables the application to manage the data transfer overheads of the tasks. The control over the binding of data is also necessary for the adaptations during operation that adjust the partitioning of the workload according to the observed conditions.

Merging outputs of tasks: The application-level control of the partitioning requires similar control over the merge operations so the outputs of tasks created from the partitions are correctly aggregated to produce the final results. This control also helps correctly estimate the overheads associated with the merge phase.

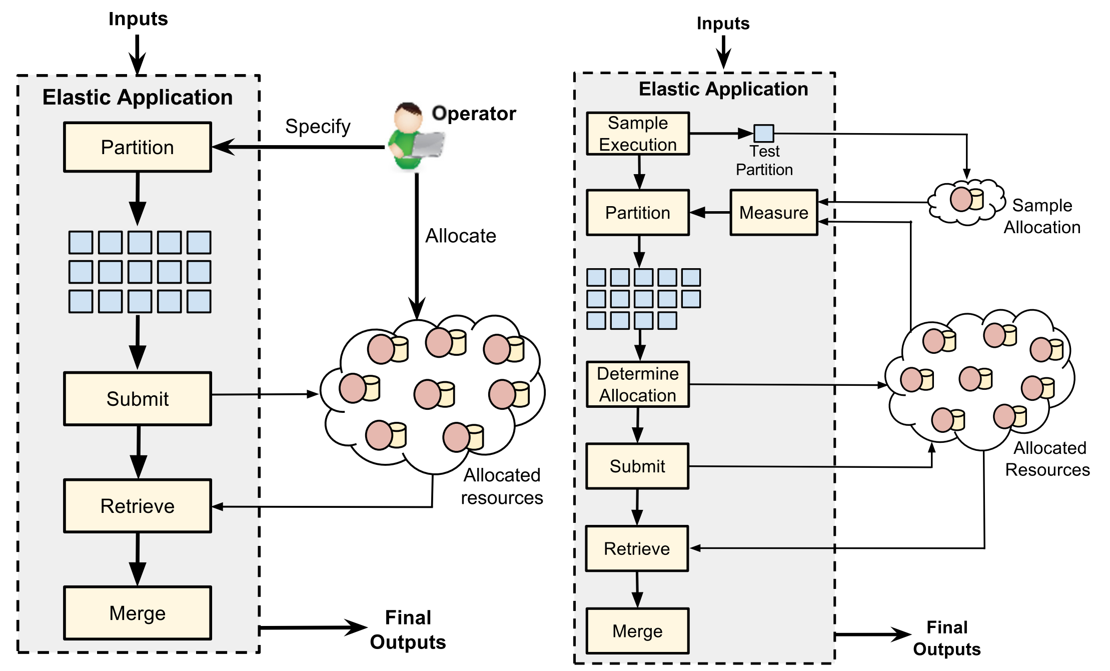
Submitting tasks for execution: The applications must direct the submission of tasks for execution and thereby, the number of simultaneous executions. This control enables applications to manage the overheads associated with transferring the common software dependencies. For example, when the transfer overheads associated with the software dependencies are large, the application can regulate the number of simultaneous executions so the transfer of these dependencies is minimized. This is because the common dependencies are cached for subsequent tasks after their initial transfer.

6.5 Application-level Adaptation

Elastic applications cannot assume, predict, or control the characteristics of the operating environment in which they are deployed. Therefore, the applications must adapt their operation to the characteristics of the deployed environment. We focus on adaptations of the two parameters that dictate the time and cost of operation: the number of partitioned tasks and the number of instances used for operation.

A simple approach for determining the number of tasks and resources to provision involves the global enforcement of default values, or requiring the operators or users to manually determine them. This approach is employed in Hadoop [53] and illustrated in Figure 6.3a. While this approach provides operators with the ability to define and control the runtime behavior, it requires detailed knowledge of the characteristics of the workload and the overheads of operation in the deployed environment. In addition, the effectiveness of this approach requires tight control over the operating environment to provide consistent characteristics throughout operation.

We present two techniques to determine the number of partitions and resources for operating the defined workload in the deployed environment without operator inter-



(a) The manual approach to partitioning using user-specified partition sizes.

(b) The sample execution-based approach that performs an assessment of the operating environment before operation.

Figure 6.3. Illustration of the current strategies for partitioning the defined workload in elastic applications.

vention. The first technique performs an initial assessment of the operating environment using a sample execution and resource allocation. It applies the measurements in the model to determine the operating parameters that achieve cost-efficient operation. This enables the optimal operation of applications in any deployed environment and operating conditions. This technique is also similar to the approach suggested by cloud providers for determining the right type of hardware and instances for running a workload [6]. Figure 6.3b illustrates this technique for adapting the operation according to the characteristics of the deployed environment. The effectiveness of this technique requires the operating conditions that impact the performance of the applications to remain unchanged during operation.

However, the operating conditions in distributed computing platforms, especially

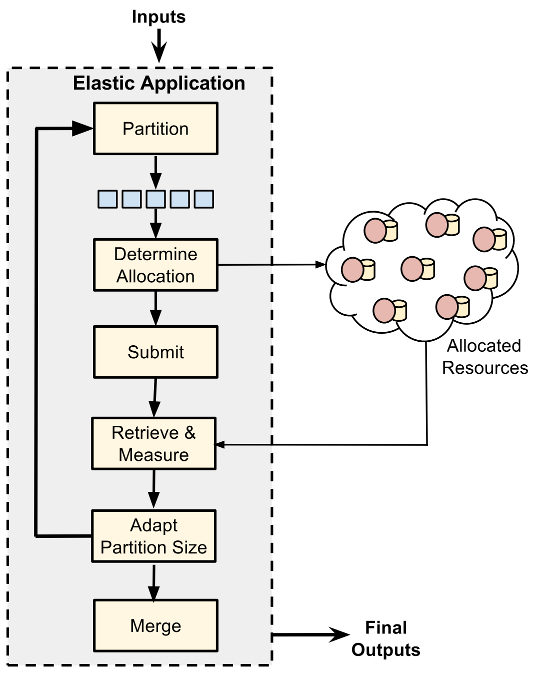


Figure 6.4. The adaptive approach that continually measures and adapts to the characteristics of the operating environment.

network bandwidth, are prone to vary during operation due to multi-tenant effects such as congestion, varying load on the shared network links, and oversubscription of the networking hardware. This impacts the cost-efficient operation of applications whose overheads of operation are influenced by these conditions. In this work, we focus on the adaptations to changes in the network bandwidth during operation.

To handle changes in the operating conditions during runtime, the second technique periodically measures the operating conditions and dynamically adapts the number of partitions and instances chosen for operation. This technique progressively partitions the workload, measures the conditions during operation of the submitted set of partitions, and applies the measurements in determining the number of partitions and instances for operating the remainder of the workload under the current conditions. The technique can also be similarly applied in applications with multiple iterations of the map operation or the split-map-join phases. Figure 6.4 illustrates this approach. While the dynamic adaptations can react to changes in the operating conditions, they can be disadvantageous when the changes in the conditions are incessant, short-lived, or prone to frequent spikes.

The operating conditions measured in both adaptation techniques include the execution overheads of the tasks, the network bandwidth and the local overheads of partition and merge. The measurements are incorporated in the model expressed in Equations 6.1, 6.8, and 6.9 to estimate the parameters that achieve cost-efficient operation of the defined workload.

The measurements on the operating conditions are obtained using API calls in Work Queue that return information on the network bandwidth, the execution time of completed tasks, and the size of the physical memory measured by the workers at the instances on which they run.

6.6 Experimental Evaluations

We apply the presented techniques in building two self-tuning applications - Elastic Sort (E-Sort) and Elastic MAKER (E-MAKER).

As we noted earlier, the merge overheads for E-Sort and E-MAKER differ due to the workload and the implementation of the merge algorithm. In E-Sort, the partitions are merged using a k-way merge algorithm that iteratively compares the records in the partitions and aggregates them in sorted order. The asymptotic running time of the algorithm is $O(N * K)$. The merge overheads of E-Sort are modeled as

$$T_{merge} = (c * N * K) + (d * N), \tag{6.10}$$

where c represents the cost of the comparisons in the merge algorithm and d is the cost associated with reading the sorted records in the partitions. On the other hand, the merge in E-MAKER is trivial since the results of the tasks - the annotated sequences - are simply concatenated in a output directory. Hence, we model the overheads of merge to be constant and negligible.

Organization: We begin our analysis of self-tuning applications by observing the effects of the number of partitions on the operating time and cost. We observe differences in the effects of the number of partitions on the overheads of partition, merge, data transfer, and task executions in E-Sort and E-MAKER. The application-level models in E-Sort and E-MAKER provide estimates on the time and cost of operation considering these effects. We experimentally validate these estimations with the goal of not showing estimations that perfectly match with the observed values, but to show the effectiveness of the model in providing information about the overheads of operation and their impact on time and cost.

The validations enable us to utilize the model in studying the effects of the characteristics of the workload and operating environment on the time and cost of operation using different partitions. The study establishes that decisions on the number of partitions and instances for cost-efficient must be made considering the characteristics of the workload and operating environment. We then experimentally demonstrate the self-tuning capabilities of E-Sort and E-MAKER in determining the number of partitions and instances to provision for cost-efficient operation. The applications utilize the application-level model and control to determine these parameters.

Our analysis considers the number of instances provisioned for operation to be equivalent to the number of partitions chosen for operation as it enables the simultaneous execution of the partitions. However, it may be useful to create smaller partitions to achieve faster failure recovery, manage resource consumption, and obtain measurements on the operation at smaller intervals. Therefore, we study the effects of *over-partitioning* where the number of partitions is greater than the instances determined for cost-efficient operation. We observe the impact of over-partitioning to be negligible when the partition and merge overheads are minimal. In such cases, over-partitioning can be useful for adapting to changing operating conditions without incurring additional overheads. We evaluate the dynamic adaptation technique that progressively over-partitions and operates the workload when the characteristics of the deployed environment, such as network bandwidth, vary during operation.

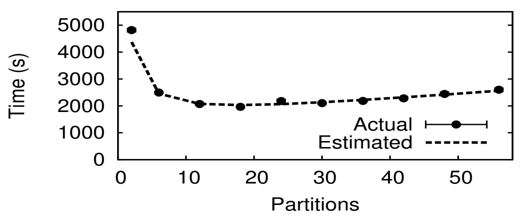
Experimental platforms and inputs: Our evaluations are done using on-demand instances from Microsoft Azure [3] - a commercial cloud platform, Future-Grid [7] - a national infrastructure that provides an IaaS testbed for building large-scale applications, and Notre Dame CRC - a campus-wide infrastructure at the University of Notre Dame that offers access to IaaS instances.

The E-Sort runs in our evaluations operate on 2 billion records that total 11GB in size. The E-MAKER runs operate on 800 contiguous sequences of the Anopheles gambiae PEST strain which amount to 7.5MB. The software overheads for E-Sort consist of the transfer of the GNU Sort executable which is 100KB. In E-MAKER, the software overheads include the reference dataset and the libraries required for the execution of the MAKER tool and are 4GB in size.

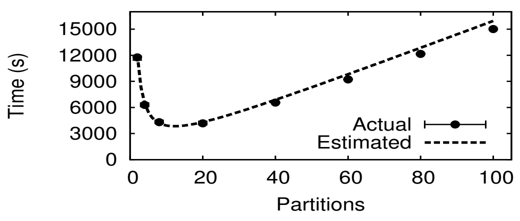
6.6.1 Validation of the estimations from the model

We validate the application-level model by comparing the estimations from Equations 6.1, 6.8, and 6.9 against the values observed in operation. Figure 6.5 presents the comparison of the estimated and actual time, cost, and cost-time product of operation for E-Sort and E-MAKER. The observed values for E-Sort were recorded during operation on Microsoft Azure instances when the network bandwidth was measured at 800 Mbps. On the other hand, E-MAKER was observed in operation on FutureGrid instances when the network bandwidth was 200 Mbps. As we described earlier, the cost of operation on these platforms is computed after rounding the time of operation to the nearest hour.

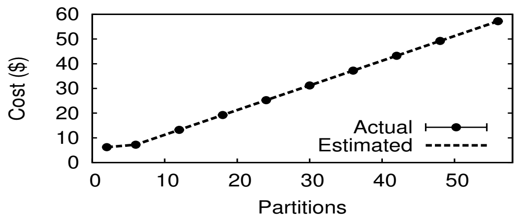
The estimations from the model use the same values for the execution time of a



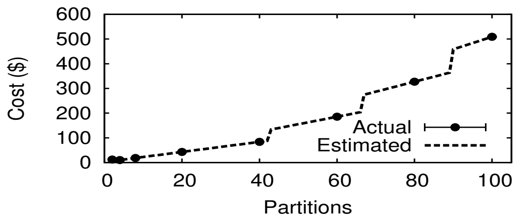
(a) $T_{operation}$: E-Sort



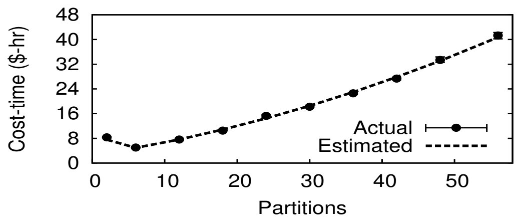
(b) $T_{operation}$: E-MAKER



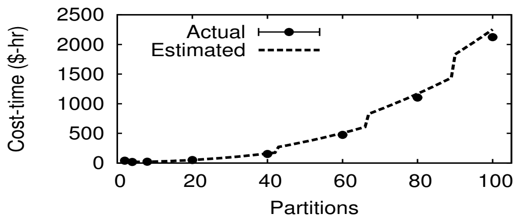
(c) $C_{\$}$: E-Sort



(d) $C_{\$}$: E-MAKER



(e) *Cost-time product*: E-Sort



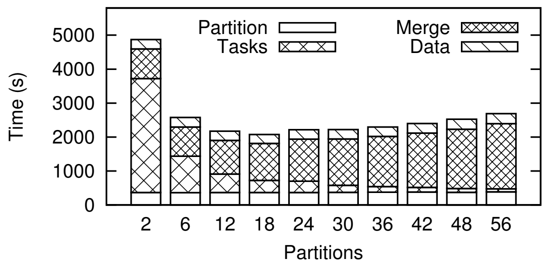
(f) *Cost-time product*: E-MAKER

Figure 6.5. Comparison of the observed values during operation with the estimations from the model for E-Sort and E-MAKER.

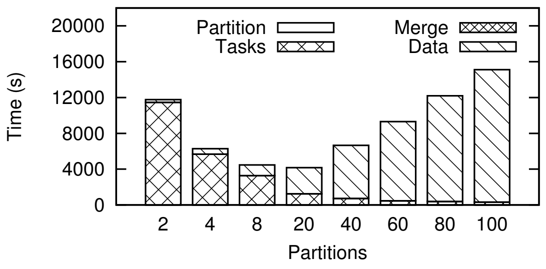
The run times in this figure were averaged over 3 runs and the error bars describe the observed minimum and maximum values.

task in Equation 6.3 and the constants in Equations 6.2 and 6.10 as the measurements made from the execution of the sample partitions by the respective applications. As before, we maintain the instances (R) for operation to be equivalent to the number of partitions (K) in Figure 6.5.

The model correctly estimates the overheads of concurrent operation and their impact on the time and cost of operation. Figure 6.5 shows the estimations from the model on the time, cost, and cost-time product of operation reflect the values observed during operation. The validation of the model enables us to use the estimations from



(a) E-Sort



(b) E-MAKER

Figure 6.6. Comparison of the overheads of the partition, task execution, data transfer, and merge operations during operation of E-Sort and E-MAKER.

the model in analyzing the operation of applications with different characteristics of the workload (such as higher task execution times), operating parameters (such as number of partitions), and operating environments (such as network bandwidth). Further, the estimations from the model can be applied in correctly identifying the optimal number of partitions and instances to provision for operation.

We experimentally observe the impact from the number of partitions on performance and break down the impact on each of the runtime components. Figure 6.6 plots the individual runtimes of the partition, task execution, merge, and data transfer components of E-Sort and E-MAKER. In this figure, the number of instances provisioned for operation is equivalent to the number of partitions.

Figure 6.6a shows the operating time of E-Sort for different partitions is dictated by the task execution times and merge overheads. The task execution times decrease exponentially while the merge overheads increase linearly as the number of partitions increases. The opposing trends in the task execution and merge overheads results in a running time that decreases with increasing partitions until the merge overheads outweigh the decrease in the task execution times.

In contrast, the operating time of E-MAKER, plotted in Figure 6.6b, is deter-

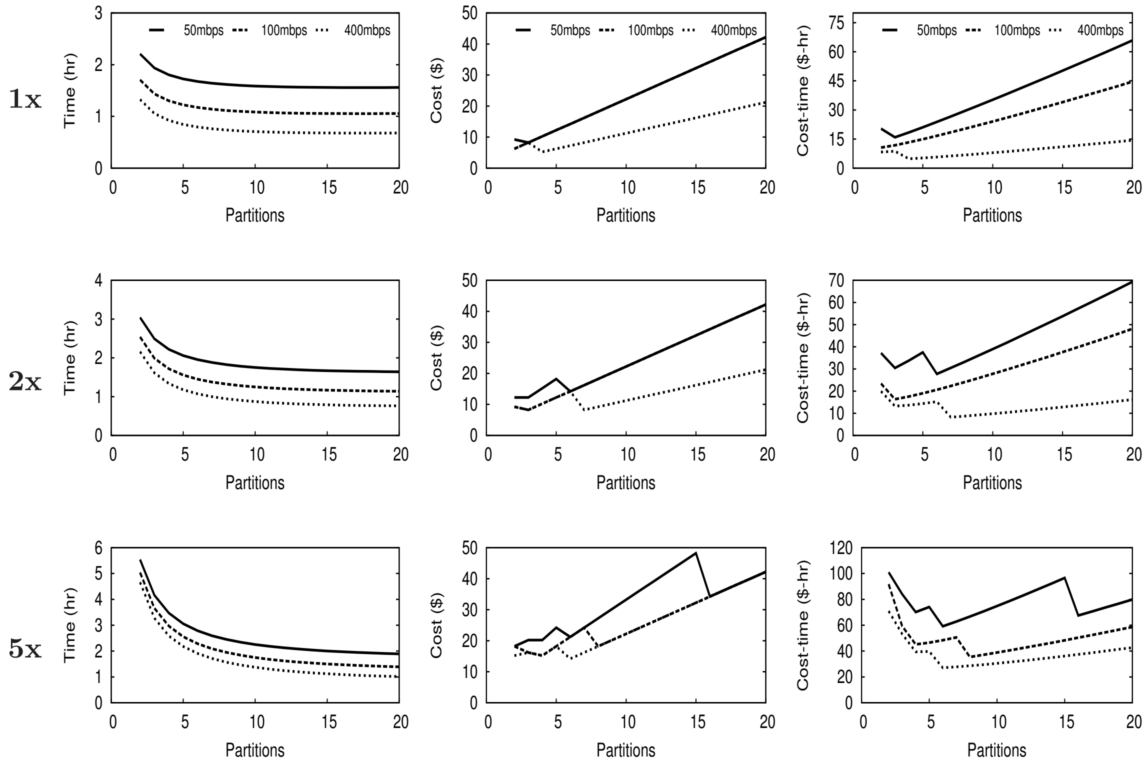


Figure 6.7: Estimated operating time, operating costs, and cost-time product of E-Sort for sorting 2 billion records totaling 11GB under various characteristics of the operating environment.

mined by the data transfer overheads and the task execution times. In E-MAKER, the data overheads increase linearly due to the software dependencies being transferred to each compute instance where the tasks are executed. As a result, the increase in the data overheads offset the gains in the concurrent executions as the partitions increase.

In summary, we observe two distinct patterns in the impact of the partitions on the operation of the two applications. The concurrency in operation is counteracted by the partition and merge overheads in E-Sort and the data overheads in E-MAKER. The model formulated in Section 6.3 provides estimations on these overheads and their impact on the time and cost of operation.

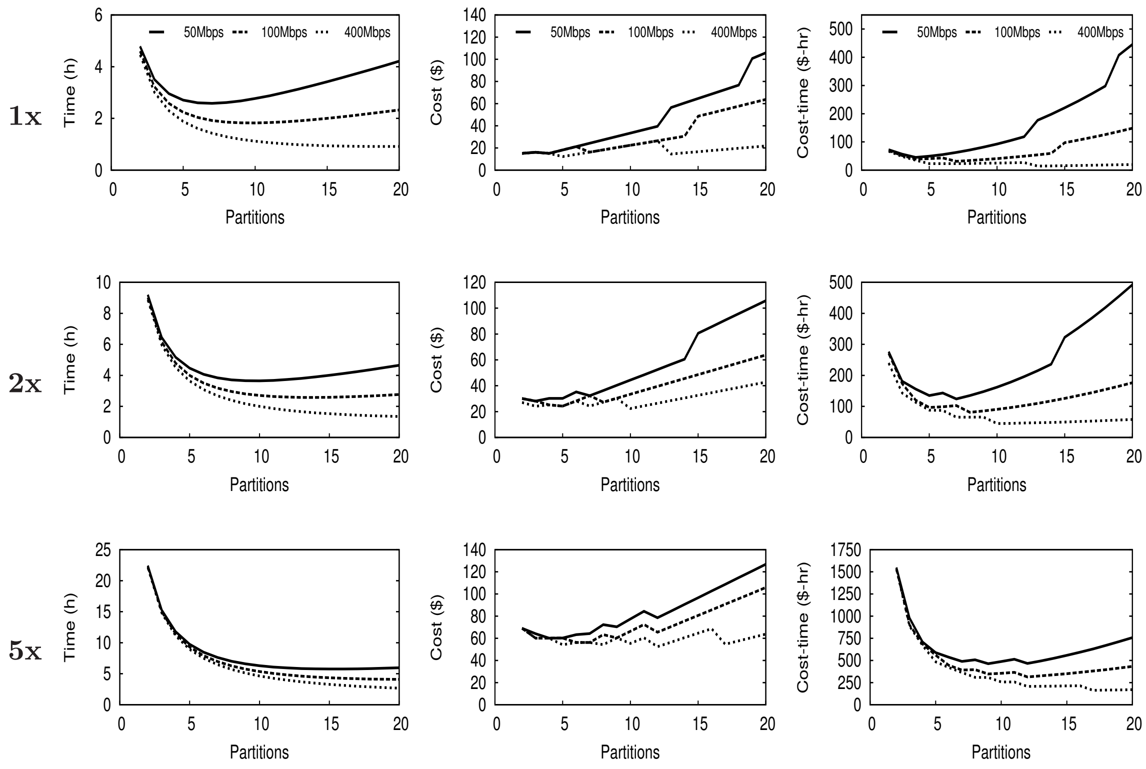


Figure 6.8: Estimated operating time, operating costs, and cost-time product of E-MAKER for annotating 800 contiguous sequences of the Anopheles gambiae genome for various characteristics of the operating environment.

6.6.2 Effects of the characteristics of the workload and operating environment

We study the choice of the number of partitions by considering the impact of the characteristics of the workload and operating environment on cost-efficient operation. In this study, we compare workloads with different task execution overheads that incur the same partition, merge, and data overheads. That is, we vary the gains in concurrency by increasing the execution times of the tasks relative to the other overheads of operation. These configurations are analogous to applications with similar but complex workloads where the task execution overheads are dominant in the overheads of operation.

We also consider operation under different network bandwidth (which impacts the transfer overheads) as it can vary between deployments due to differences in the platform configurations and hardware. Further, the network resources are often shared among multiple tenants of IaaS and PaaS platforms resulting in variations of the bandwidth from traffic patterns, congestion, and demand for resources.

Figures 6.7 and 6.8 illustrate the effects of the characteristics of the workload and network bandwidth on the operating time, costs, and cost-time product for various partitions in E-Sort and E-MAKER respectively. In these figures, each row plots the operation for different execution times of the tasks varied relative to those observed in Figure 6.6. The overheads of partition and merge operations are the same as plotted in Figure 6.6.

The increase in the execution time of the tasks relative to the partition, merge, and data overheads leads to an increase in the gains realized from concurrent operation. Therefore, the number of partitions that achieve lower operating times increases with higher task execution times as observed in Figures 6.7 and 6.8. Similarly, an increase in network bandwidth lowers the data transfer overheads and increases the gains realized from operation with higher number of partitions. This effect is seen in Figure 6.8 for E-MAKER due to the large transfer overheads from software dependencies.

As we noted in Section 6.2, the operating costs exhibit irregular trends due to the rounding of the operating time to the nearest hour. That is, the cost and cost-time product of operation with increasing partitions in Figures 6.7 and 6.8 are influenced by the magnitude of the decrease in the operating time and if the decrease results in a drop to the next lowest hourly boundary.

In summary, the determination of the number of partitions and instances to provision must be made in conjunction with measurements of the characteristics of the workload and operating environment.

6.6.3 Adaptations to the characteristics of the workload and operating environment

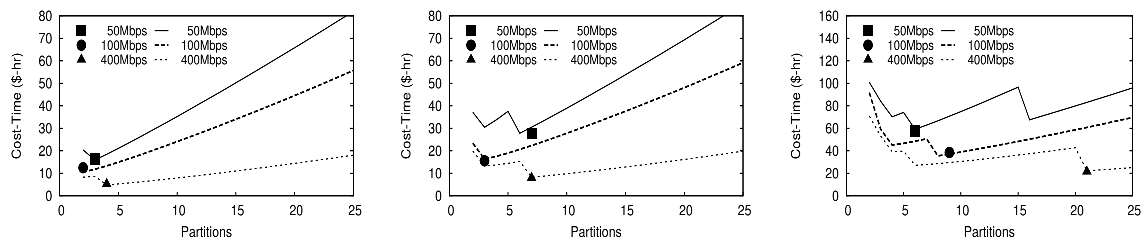
In this section, we show the application-level adaptations of the number of partitions and instances used in operation based on the initial assessment of the overheads of operating the workload in the deployed environment. The adaptations measure the characteristics of the workload (such as task execution times) and operating environment (such as network bandwidth) by operating a sample partition on a sample allocation. The sample partition comprises about 1% of the defined workload in the applications. The sample allocation consists of a single instance in the same environment in which the application will be operated. This sample allocation is further used in operating the remainder of the workload to prevent wastage as instances in metered platforms incur charges to the nearest hour.

The measurements using the sample partition and allocation provide information on the network bandwidth and the overheads of task execution, partition, and merge. Based on these measurements, the applications determine the number of partitions and instances to provision for cost-efficient operation using estimates from the model.

Figures 6.9 and 6.10 show the chosen partitions along with the actual cost-time product observed when operating with those partitions. The overheads of operating the sample partitions are minimal compared to the overall time of operation. This can be observed in Figures 6.9 and 6.10 where the actual cost-time product closely tracks the cost-time product estimated by the model. In summary, these figures show that the applications achieve cost-efficiency by tuning their operation to the operating conditions in the deployed environment.

6.6.4 Over-partitioning of workload

Our analysis so far considers the number of partitions and instances provisioned for operation to be equivalent. The over-provisioning of instances relative to the number of partitions chosen for operation results in resource wastage and high costs. In



(a) T_{task}: 1x (b) T_{task}: 2x (c) T_{task}: 5x

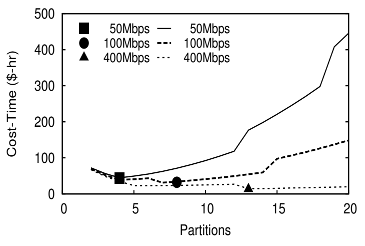
Figure 6.9: Illustration of the partitions determined for sorting 2 billion records totaling 11GB using the application-level model and measurements of the operating environment.
The lines represent the estimated values for sorting 2 billion records under different bandwidth. The individual points plot the partitions dynamically chosen during operation by measuring the operating environment.

this section, we consider the over-partitioning of the workload relative to the number of instances provisioned for operation. The over-partitioning is useful when smaller partitions or tasks are desired for faster detection and re-execution of failed tasks, limiting the consumption of resources by tasks, and quickly adapting the operation to varying operating conditions measured from the execution of tasks.

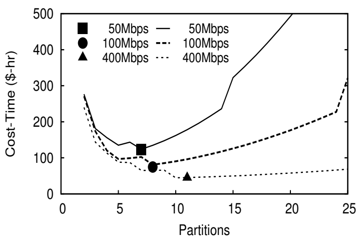
Figure 6.11 describes the effects of creating partitions greater than the number of instances determined for cost-efficient operation. The over-partition factor represents the multiplicative factor applied on the number of instances determined for cost-efficient operation.

In E-Sort, over-partitioning incurs higher partition and merge overheads without recording any increase in the gains from the increased concurrency. This is because the number of partitions that can be simultaneously executed is limited by the number of instances available for operation. As a result, the time and cost of operation of E-Sort increases with over-partitioning as seen in Figure 6.11a (at over-partition factor of 6, the time of operation increases to the next hourly boundary resulting in a sharp increase in the cost-time product).

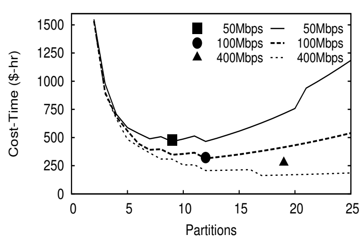
In E-MAKER, the effects of over-partitioning on the time and cost of operation



(a) T_{task}: 1x



(b) T_{task}: 2x



(c) T_{task}: 5x

Figure 6.10: Illustration of the partitions determined for annotating 800 sequences of Anopheles gambiae using the application-level model and measurements of the operating environment.
The lines represent the estimated values for annotating the sequences under different bandwidth. The individual points plot the partitions dynamically chosen during operation by measuring the operating environment.

are marginal. This is because the partition and merge overheads are negligible and the data transfer overheads only increase with the number of instances used for execution of the tasks. In the next section, we utilize over-partitioning in E-MAKER to enable the measurement of varying operating conditions at shorter intervals and the adaptation of the operating parameters to the measured conditions.

6.6.5 Adaptations to varying operating conditions

In this section, we demonstrate the dynamic adaptations when the characteristics of the operating environment that impact the overheads and performance of the applications vary during operation. We consider changes in the network bandwidth since it is prone to vary due to multi-tenant effects. We show the dynamic adaptations in E-MAKER where the impact from changes in the bandwidth are pronounced due to the large common data transfer overheads.

Figure 6.12 shows the dynamic adaptations by E-MAKER to the operating conditions observed during runtime. It plots the number of partitions and instances chosen for operation based on the observed bandwidth. The adaptations in E-MAKER operate by progressively partitioning and allocating the instances for operation based

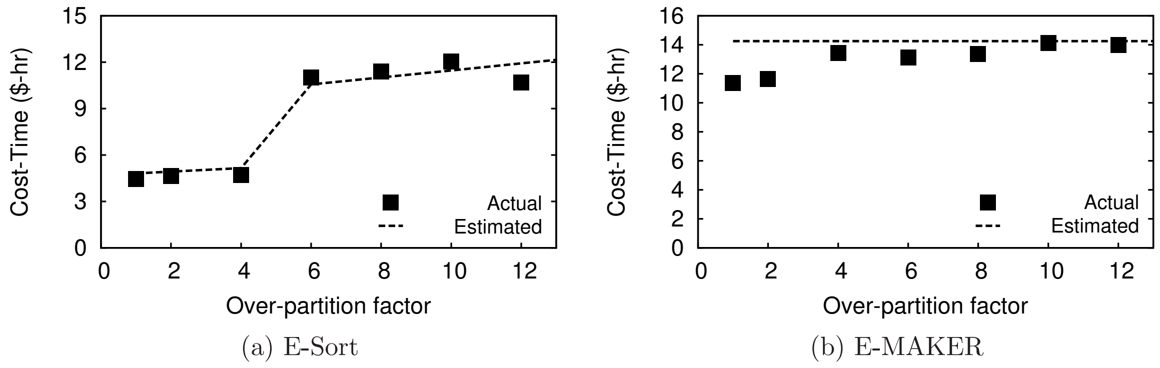


Figure 6.11. Illustration of the effects of over-partitioning in E-Sort and E-MAKER.

The actual values were observed with the same experimental setup and inputs as Figure 6.5.

on the observed conditions. It also over-partitions the workload by a factor of 2 to enable measurement and adaptation at shorter intervals without incurring additional overheads. In our setup, E-MAKER measures the operating environment after the dispatch of every task and recomputes the operating parameters.

We observe in Figure 6.12 that when the bandwidth drops after 300 seconds of operation, E-MAKER recomputes its operating parameters and lowers the number of partitions and instances it uses for operation. This minimizes the transfer overheads which become pronounced at low bandwidth. When the bandwidth increases again at 2100 seconds, E-MAKER determines that it can achieve cost-efficiency by continuing operation with the current scale of instances rather than increasing the instances in the deployment. In the experiment show in Figure 6.12, the overheads of progressive partitioning and re-computation of the operating parameters was less than 5% of the time of operation.

For comparison, Figure 6.12 also plots the operation of E-MAKER using the measurements from the operation of sample partitions in the same operating environment. From the comparisons of the two techniques, we note the dynamic adaptations are better positioned to handle variations in the operating environment while incurring low overheads. However, the dynamic adaptations can incur large overheads and prove disadvantageous when the variations are spurious and frequent.

6.7 Conclusion

We considered the deployment and operation of concurrent applications in distributed computing platforms using resources exclusively provisioned for each instance of the applications. The cost-efficient operation in these environments is determined by the number of partitions and compute instances chosen for operation. We show the scale of partitions and instances that achieve cost-efficient operation vary significantly depending on the characteristics of the workload and the environment in which they are operated. Further, these operating parameters must be correctly determined in diverse, unknown, and often unpredictable operating environments in which the applications are deployed. In order to determine the number of partitions and instances for cost-efficient operation in the deployed environment, we argue that applications must be self-modeling and self-tuning. In this chapter, we considered the class of applications operated using the split-map-merge paradigm and demonstrated application-level techniques for realizing self-modeling and self-tuning applications. In addition to enabling time- and cost-efficient operation, self-tuning applications also provide the following benefits to their operators: (1) relieve them from the burden of determining and maintaining the operating environment that enables efficient operation, and (2) enable them to deploy on resources in any distributed computing platform accessible to them.

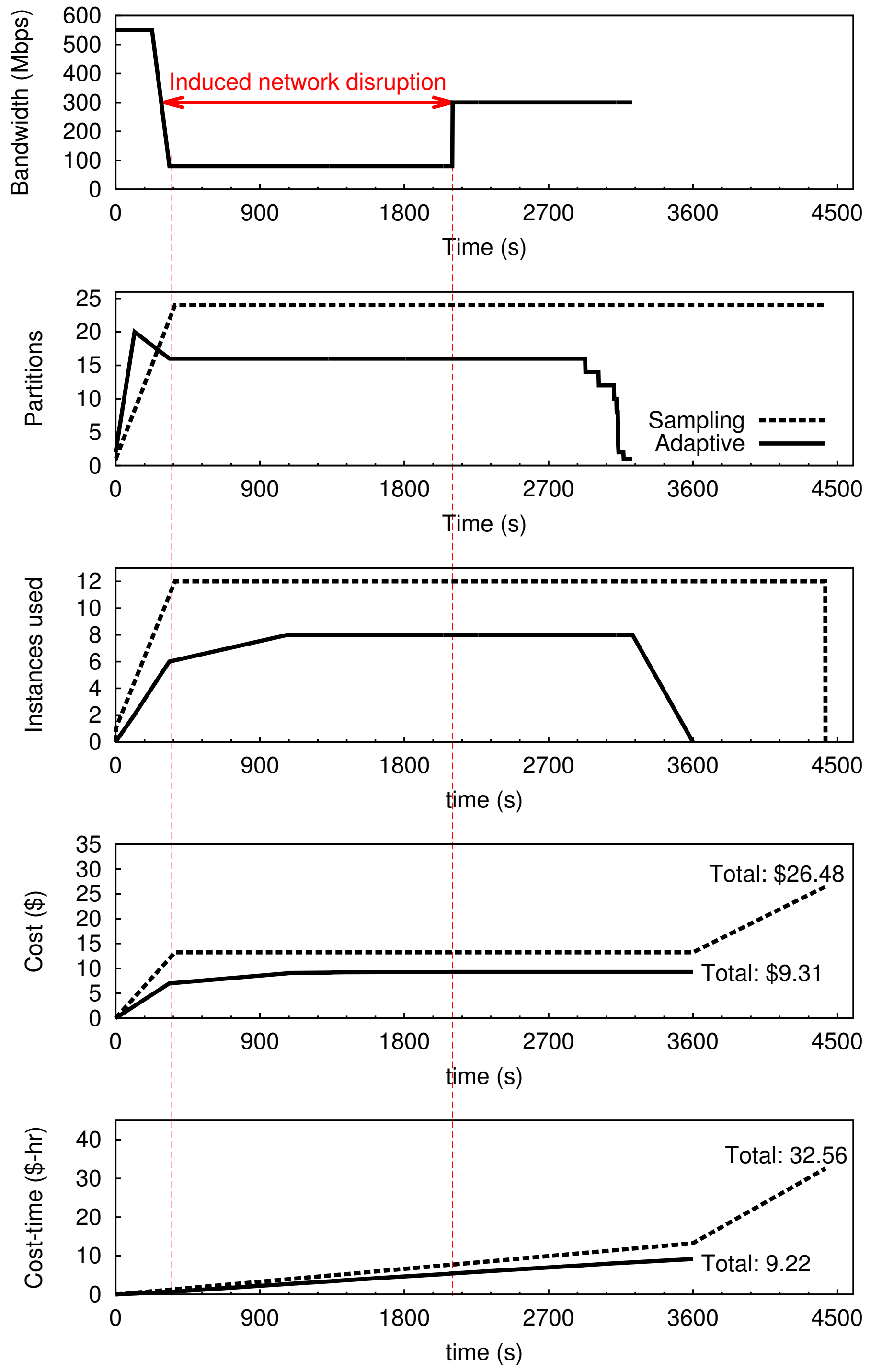


Figure 6.12. Dynamic adaptations of the operating parameters in E-MAKER according to the observed network bandwidth.

For comparison, the operation using the parameters chosen from the initial sampling of the environment is also plotted.

CHAPTER 7

CONCLUSION

7.1 Recapitulation

The current and future generations of concurrent scientific applications will increasingly be deployed on-demand by their end-users on currently accessible, federated, and heterogeneous pools of resources. As a result, the platform, hardware, and operating environment of the applications will vary from one user to another and from one invocation to another. To efficiently operate in such scenarios where the operating environment is unknown and prone to vary, I argued and demonstrated the need for elastic applications.

An elastic application can execute with an arbitrary and varying set of available resources during runtime. These applications can seamlessly adapt their execution to operate with the currently available resources. That is, elastic applications can operate without any restriction on the size, reliability, hardware, and performance characteristics of their resource allocation. The end-users can deploy elastic applications on any infrastructure or platform of choice, and allocate and remove resources during execution based on their needs and constraints.

I presented an in-depth study of six elastic applications in terms of their construction, operation, and overheads. The experiences and observations in building the above applications were distilled into six high-level guidelines: *(1) Abolish shared writes, (2) Keep your software close and your dependencies closer, (3) Synchronize two, you make company; synchronize three, you make a crowd, (4) Make tasks of a* *feather flock together, (5) Seek simplicity, and gain power, and (6) Build a model before scaling new heights.*

Next, I showed that the direct deployment and operation of elastic applications on the current generation of distributed computing systems such as clouds incurs monetary costs to the end-users. Elastic applications therefore must achieve cost-efficient operation in the environment deployed by their end-users. The cost-efficient operation in these environments is determined by the number of partitions and compute instances provisioned for operation. I also showed that the scale of partitions and instances that achieve cost-efficient operation vary significantly depending on the characteristics of the workload and the environment in which they are operated.

I started with techniques in the middleware for improving the efficient operation of elastic applications. I presented two key techniques in the middleware that (1) exposed selective information about the operating environment (such as the network bandwidth) to the applications, and (2) utilized a hierarchical data strategy for distributing the data to the resources provisioned for concurrent execution of the application. The first technique defies conventional wisdom and shows applications can benefit from key information about the underlying execution environment when determining the parameters and resources for operation. The second technique argues for better data transfer and management mechanisms in the middleware in order to lower the overheads and costs of operation on distributed computing systems.

I continued the search for techniques for achieving efficient operation by moving my focus to the application layer. The applications hold fine-grained knowledge of the characteristics and the current and future needs of the workloads they run. To effectively harness and apply this knowledge towards improving the efficiency of applications, I proposed and demonstrated three techniques. First, the applications must be self-modeling. The applications must incorporate a model of their workload characteristics, structure, and operating parameters that influence their execution. Second, the applications must explicitly control and direct the expression and operation of the concurrent units of their workload. Third, the applications must implement self-tuning mechanisms that adapt their behavior and operation during runtime according to the characteristics of their deployed environment.

Finally, I provided evaluations of self-operating elastic applications built using the proposed techniques. The evaluations showed that self-operating applications achieve high efficiency in terms of their running time and monetary costs incurred during operation despite the overheads. This is due to the applications being able to make effective decisions on their operating parameters during execution using measurements of the operating conditions during runtime.

7.2 Operators, Agents, and Abstractions

Members of my research group have successfully argued and shown that the complexities of operation on distributed systems are best addressed by agency [110] and abstractions [81]. The agents shield applications from having to navigate the intricate web of different interfaces, protocols, and services in various distributed systems. They handle the negotiation of access and usage of various resources in the distributed systems and coordinate the functions of multiple components (e.g., remote storage, data transfer) to provide a seamless and productive operation of the applications. Abstractions, on the other hand, help applications express and execute their workload without having to deal with the heterogeneities in the hardware, performance variations, and failures prevalent in distributed systems. They simplify the construction of applications and enable their developers to focus in the design of algorithms to operate on the inputs without having to worry about the semantics of operation. In summary, agency and abstractions enable the applications focus on executing their workload correctly and efficiently without concern for the underlying details of their operating environment.

In the ecosystem comprised of agents and abstractions, my work defines and casts the role of operators. It shows applications as the operators who determine and direct the decisions for efficient operation. The self-operating applications seek and apply the functions offered by the agents and abstractions for their construction and operation on distributed computing systems.

This model is not unlike the system found in modern society where human beings employ agents (e.g., travel agent, financial agent) for handling the complexities and the details of carrying out an operation (e.g., traveling to a destination, investing money). The human beings at the same time also use various abstractions (e.g., airlines, banks) to perform the operations or accomplish the desired work. The agents and abstractions enable human beings to navigate the complex tasks and interfaces (e.g., finding available flights with favorable times, finding investment options with the high risk-to-reward ratio). However, in the end, human beings maintain control over the work carried out by the agents and abstractions on their behalf, make the decisions regarding the work being performed, and act on the decisions at their discretion in a manner that suits their circumstances and goals. This dissertation can be considered to extend this analogy to software applications operating on distributed computing systems and demonstrate it in practice.

7.3 Impact

The application-level techniques presented in this work can be considered to extend the well-known end-to-end principle of system design [97] to large-scale distributed computing. The presented techniques reside at the end-points (applications) of the distributed computing software stack and enable cost-efficient decisions during operation.

The application-level model introduced and described in this work is being utilized in accurately predicting the resource consumption of application and the configuration of the resources for the operation of large data-parallel scientific applications. The model is applied in building mathematical techniques that find optimal values for multiple variables such as the number of instances, size and type of instances, and the number of partitions.

In addition to the presentation of the principles for the design and operation of large-scale concurrent applications, this dissertation also included the construction of applications that were used actively in scientific work in their respective fields.

This work extended Elastic Maker [112] by incorporating a model of the runtime components and dynamic adaptation mechanisms for self-operation. The Elastic Replica Exchange was used in biomolecular research at Notre Dame for examining the movements of protein molecules across their conformational spaces at a finer granularity. The feedback from their use showed the replica exchange to be slow and severely affected by the presence of heterogeneous hardware and failures due to the use of the global synchronization barriers. This resulted in the work on removing the global barrier and improving the runtime performance in the presence of heterogeneous hardware and failures.

The Accelerated Weighted Ensemble (AWE) was built as part of a collaborative effort with scientists involved in biomolecular research at Notre Dame. This work extended AWE to use a hierarchical data distribution model enabling it to scale using federated resources. This setup also minimized the impact from failures and lowered the data transfer overheads. Finally, the Elastic Sort implementation was useful to the broader community as a model implementation highlighting the components, benefits, and overheads of elastic applications.

All the elastic applications presented in this work are now shipped as part of the CCTools software package [4] with the exception of AWE which is shipped as a standalone product [2].

7.3.1 Publications

The characteristics of elastic applications and the conversion of a concurrent scientific application to an elastic application was published at IEEE CloudCom 2012 [92]. The case studies of elastic applications and the lessons learned from their construction and operation were published at IEEE CCGrid 2013 [93]. The demonstration and evaluation of the hierarchical data distribution model in the operation of AWE was presented at IEEE Cluster 2013 [13]. The techniques for self-operating applications, namely application-level modeling, control, and adaptations, were presented in a journal paper that was accepted for publication in the IEEE Transactions on Cloud Computing in January 2015.

7.4 Future Work

The ideas and techniques presented in this work can be considered to establish important and useful steps in the realization of fully autonomous distributed computing applications operating at scale using thousands of resources spread across multiple platforms. In other words, this work makes contributions towards a future where applications fully control and direct their operation in the deployed environment without requiring any effort from their operators. These applications, when invoked, automatically determine the execution parameters, provision the resources that will enable optimal operation of their workloads, and adapt to any changes in the deployed environment that impacts prior estimations on efficient operation. To offset the changes in the operating conditions, the applications will alter their execution as well as the size, type, and scale of the allocated resources. Further, the applications will let the end-users specify their desired operational goal for the current invocation in terms of optimal time, cost, or both and allow these goals to be modified runtime while incurring a minimal penalty. To fully realize this goal of autonomous distributed computing, the following work remains to be accomplished.

7.4.1 Application-level techniques

The applicability and usefulness of the techniques presented in this work are currently limited to applications that conform to the split-map-merge execution paradigm. Applications with workloads expressed in the form of Directed Acyclic Graphs (DAG) are growing in their size and popularity [62, 84, 109] and might benefit largely from the extension of the techniques presented in this work. The earlier work in Pegasus [42] presented techniques for the partitioning of the DAG to achieve optimal scheduling and mapping of the computations to available resources. In a similar way, techniques must be established for the partitioning or aggregation of the nodes expressed in the DAG in order to achieve time- and cost-efficient operation.

It might also be worthwhile to build techniques and solutions that determine the size of the data or computation that must be distributed for remote execution. As the capacities and capabilities of resources at an individual execution site continue to grow, it might prove efficient to leverage the local resources at the master-coordinator to run a piece of the computation. Further, the pipelining of the local execution with the data transfer and execution of the tasks on the allocated resources may result in substantial gains in efficiency.

Finally, the partitioning of the workloads can be improved by considering the failure rates and the probability of the termination of instances in grids and spot-price instances in cloud platforms. These events are common and expected in these platforms due to their operating principles and business models. The failures and premature terminations of resources can impact the performance and efficiency of the applications with a certain partition size since the cost of migrating and re-executing a partition increases with the size of the partition.

7.4.2 Resource selection

The efficient operation of applications can be improved by considering the size (e.g., large vs small instances in cloud platforms) and type (e.g., dedicated- vs spot-instances in cloud platforms) of instances to provision for their operation. It might be useful to start by considering the operation of the individual partitioned tasks as multi-threaded processes at the execution sites. This work showed the trade-offs that exist in the overheads between the partitioning of the workload into multiple tasks and the operation of each task as multi-threaded programs on multiple cores. Techniques that determine the partitioning and configuration of tasks and threads where the overheads are minimal are required. These techniques can then be extended to also compute the number of instances and the capacities of the resources (cores, memory, disk) to provision at each instance.

Recently, hardware accelerators (e.g., Intel Xeon processors) and GPUs have been adopted in the operation of large scale and resource intensive computations [70, 118]. The overheads of partitioning, data movement, and execution on these hardware accelerators need to studied and evaluated. Further, the differences with their CPU counterparts need to be modeled so they can be applied in operating environments that include a mix of CPU, GPU, and hardware accelerators in the operation of applications such as AWE.

Finally, the federation of resources across multiple platforms presents challenges to the estimation and selection of resources. The use of the hierarchical configuration of masters in the form of foremen described in Chapter 5.4 was a good first step in managing resources derived from multiple platforms. However, challenges still remain unaddressed on how applications should determine the size, type, and scale of resources to provision in each platform. Further, it is also important for applications to account for the differences in the characteristics and performance of the resources and isolate their effects.

7.4.3 Cost optimization

The cost-efficient operation of applications requires further sophistication in the techniques at the application layer. An useful sophistication of the presented techniques will provide the capability to switch to a different platform, or a different type or size of instances if the operating conditions vary enough to warrant such a migration during execution. Another direction will consider terminating and migrating the currently running computations to newly allocated instances that can complete the computations at a cheaper cost or at a faster rate.

Such sophistication during runtime requires mechanisms that can communicate and interface with the applications to allocate and manage resources based on the requirements communicated by the applications. The Work Queue Pool tool [126] developed by my research group is a step in this direction. This tool can be extended to terminate instances when the applications do not find them useful during runtime, allocate new instances that the applications determine to be necessary for achieving efficient operation, and manage the resources at the desired scales especially in the presence of failures and terminations.

BIBLIOGRAPHY

1. Cygwin https://www.cygwin.com, accessed April 2015.

2. Accelerated Weighted Ensemble Python Library. `https://github.com/cooperative-computing-lab/awe`. Accessed: 2015-02.

3. Windows Azure Cloud Platform. `http://www.windowsazure.com`. Accessed: 2013-12-21.

4. Cooperative Computing Tools. `https://github.com/cooperative-computing-lab/cctools`. Accessed: 2015-02.

5. The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman, 2002.

6. How do I select the right instance type? `http://aws.amazon.com/ec2/faqs`. Accessed: 2014-07-21.

7. FutureGrid. `https://portal.futuregrid.org`. Accessed: 2013-12-21.

8. B. Abdul-Wahid, L. Yu, D. Rajan, H. Feng, E. Darve, D. Thain, and J. A. Izaguirre. Folding Proteins at 500 ns/hour with Work Queue. In *8th IEEE International Conference on eScience (eScience 2012)*, 2012.

9. V. Adve, V. V. Lam, and B. Ensink. Language and compiler support for adaptive distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, LCTES '01, pages 238–246, New York, NY, USA, 2001. ACM. ISBN 1-58113-425-8. doi: 10.1145/384197.384229. URL `http://doi.acm.org/10.1145/384197.384229`.

10. S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 21–21, Berkeley, CA, USA, 2012.

11. A. Al-bar and I. Wakeman. A survey of adaptive applications in mobile computing. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 246–251, Apr 2001. doi: 10.1109/CDCS.2001.918713.

12. K. Al-Tawil and C. A. Moritz. Performance modeling and evaluation of mpi. *Journal of Parallel and Distributed Computing*, 61(2):202 – 223, 2001.

13. M. Albrecht, D. Rajan, and D. Thain. Making Work Queue Cluster-Friendly for Data Intensive Scientific Applications . In *IEEE International Conference on Cluster Computing*, 2013.

14. F. Almeida Morais, F. Vilar Brasileiro, R. Vigolvino Lopes, R. Araujo Santos, W. Satterfield, and L. Rosa. Autoflex: Service agnostic auto-scaling framework for iaas deployment models. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 42–49, 2013. doi: 10.1109/CCGrid.2013.74.

15. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 3(215):403–410, Oct 1990.

16. Amazon EC2. http://aws.amazon.com/ec2, 2014.

17. Amazon EC2 Spot Instances. http://aws.amazon.com/ec2/spot-instances, 2012.

18. Amazon.com. Elastic compute cloud (ec2). http://www.aws.amazon.com/ec2.

19. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967. doi: 10.1145/1465482.1465560.

20. D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2256-4.

21. A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, 1998.

22. J. Basney, R. Raman, and M. Livny. High Throughput Monte Carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

23. T. Bicer, D. Chiu, and G. Agrawal. Time and Cost Sensitive Data-Intensive Computing on Hybrid Clouds. In *Cluster, Cloud and Grid Computing, 12th IEEE/ACM International Symposium on*, pages 636–643, May 2012. ISBN 978-1-4673-1395-7. doi: 10.1109/ccgrid.2012.95.

24. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN Notices*, volume 30, August 1995.

25. P. Brenner, C. R. Sweet, D. VonHandorf, and J. A. Izaguirre. Accelerating the replica exchange method through an efficient all-pairs exchange. *Journal of Chemical Physics*, 126:074103, February 2007.

26. P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

27. P. Bui, L. Yu, A. Thrasher, R. Carmichael, I. Lanc, P. Donnelly, and D. Thain. Scripting distributed scientific workflows using Weaver. *Concurrency and Computation: Practice and Experience*, 2011.

28. M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, DEC Systems Research Center, 1994.

29. R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational Grid. In *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region*, pages 283–289, 2000.

30. M. Cannataro, D. Talia, and P. K. Srimani. Parallel data intensive computing in scientific and commercial applications. *Parallel Computing*, 28(5):673–704, 2002.

31. E. Cantu-Paz. Designing Efficient Master-Slave Parallel Genetic Algorithms. pages 455+, 1998.

32. H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the Grid. *Sci. Program.*, 8(3):111–126, Aug. 2000. ISSN 1058-9244.

33. F. Chang, J. Ren, and R. Viswanathan. Optimal Resource Allocation in Clouds. In *IEEE 3rd International Conference on Cloud Computing*, pages 418–425. IEEE, 2010. ISBN 978-1-4244-8207-8. doi: 10.1109/cloud.2010.38.

34. W. Chen and E. Deelman. Integration of Workflow Partitioning and Resource Provisioning. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 764–768, 2012. ISBN 978-1-4673-1395-7. doi: 10.1109/ccgrid.2012.57.

35. W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: the mygrid approach. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 407–416, Oct 2003. doi: 10.1109/ICPP.2003.1240605.

36. D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. Scatter/gather: A cluster-based approach to browsing large document collections. In *Proceedings of the 15th International ACM SIGIR conference on Research and development in information retrieval*, pages 318–329, 1992.

37. M. D. de Assunçao and R. Buyya. Performance analysis of allocation policies for interGrid resource provisioning. *Inf. Softw. Technol.*, 51(1):42–55, Jan. 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2008.09.013.

38. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

39. J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, Jan. 2010. ISSN 0001-0782. doi: 10.1145/1629175.1629198.

40. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, 2004.

41. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281.

42. E. Deelman, G. Singh, M. H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005. ISSN 1058-9244.

43. J. J. Dongarra and D. W. Walker. MPI: A standard message passing interface. *Supercomputer*, pages 56–68, January 1996.

44. R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

45. T. N. Duong, X. Li, and R. S. Goh. A Framework for Dynamic Resource Provisioning and Adaptation in IaaS Clouds. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 312–319. IEEE, Nov. 2011. ISBN 978-1-4673-0090-2. doi: 10.1109/cloudcom.2011.49.

46. I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *Quality of Service, 2000. IWQOS. 2000 Eighth International Workshop on*, pages 181–188, 2000. doi: 10.1109/IWQOS.2000.847954.

47. I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, 2008.

48. S. Genaud and J. Gossa. Cost-Wait Trade-Offs in Client-Side Resource Provisioning with Elastic Clouds. In *IEEE 4th International Conference on Cloud Computing*, pages 1–8, July 2011. ISBN 978-1-4577-0836-7. doi: 10.1109/cloud.2011.23.

49. W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 35–, 2001. ISBN 0-7695-1010-8.

50. A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

51. A. Gounaris, N. W. Paton, A. A. Fernandes, and R. Sakellariou. Adaptive query processing: A survey. 2405:11–25, 2002. doi: 10.1007/3-540-45495-0_2.

52. J. P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, HPDC '00, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0783-2.

53. Hadoop. http://hadoop.apache.org/, 2007.

54. M. K. Hedayat, W. Cai, S. J. Turner, and S. Shahand. Distributed Execution of Workflow Using Parallel Partitioning. In *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 106–112. IEEE, 2009. ISBN 978-0-7695-3747-4. doi: 10.1109/ispa.2009.96.

55. R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.

56. T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey. FlexPRICE: Flexible Provisioning of Resources in a Cloud Environment. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 83–90. IEEE, July 2010. ISBN 978-1-4244-8207-8. doi: 10.1109/cloud.2010.71.

57. E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. In *GRID*, pages 214–227, 2000.

58. Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of windows azure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 367–376, 2010.

59. C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. On the Use of Cloud Computing for Scientific Workflows. In *eScience, 2008. eScience &#039;08. IEEE Fourth International Conference on*, pages 640–645, Washington, DC, USA, Dec. 2008. IEEE. ISBN 978-1-4244-3380-3. doi: 10.1109/escience.2008.167.

60. C. Holt and M. Yandell. MAKER2: an annotation pipeline and genome-database management tool for second-generation genome projects. *BMC Bioinformatics*, (12):491, 2011.

61. S. E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. Database Syst.*, 14(3):291–321, Sept. 1989. ISSN 0362-5915. doi: 10.1145/68012.68013.

62. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.

63. D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *7th Workshop on Job Scheduling Strategies for Parallel Processing*, 2001.

64. S. Jha, D. S. Katz, M. Parashar, O. Rana, and J. Weissman. Critical perspectives on large-scale distributed applications and production Grids. In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 1–8. IEEE, Oct. 2009. ISBN 978-1-4244-5148-7. doi: 10.1109/grid.2009.5353064.

65. S. Jha, Y. E. Khamra, and J. Kim. Developing Scientific Applications with Loosely-Coupled Sub-tasks. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 641–650, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-01969-2. doi: 10.1007/978-3-642-01970-8_63.

66. G. Juve and E. Deelman. Resource Provisioning Options for Large-Scale Scientific Workflows. In *2008 IEEE Fourth International Conference on eScience*, pages 608–613. IEEE, Dec. 2008. ISBN 978-1-4244-3380-3. doi: 10.1109/escience.2008.160.

67. Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, page 13, Berkeley, CA, USA, 2011. USENIX Association.

68. D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. Parallel supercomputing today and the cedar approach. *Science*, 231(4741):967–974, 1986.

69. B. Langmead and S. L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nat. Methods*, 9(4):357–359, Apr. 2012. ISSN 1548-7091. doi: 10.1038/nmeth.1923.

70. A. Leist, D. P. Playne, and K. A. Hawick. Exploiting graphical processing units for data-parallel scientific applications. *Concurrency and Computation: Practice and Experience*, 21(18):2400–2437, 2009. ISSN 1532-0634. doi: 10.1002/cpe.1462. URL `http://dx.doi.org/10.1002/cpe.1462`.

71. E. Lindahl, B. Hess, and D. van der Spoel. Gromacs 3.0: a package for molecular simulation and trajectory analysis. *Journal of Molecular Modeling*, 7:306–317, 2001.

72. M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Eighth International Conference of Distributed Computing Systems*, June 1988.

73. W. Lu, J. Jackson, and R. Barga. AzureBlast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 413–420. ACM, 2010. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851537.

74. A. Luckow and et al. Distributed replica-exchange simulations on production environments using saga and migol. In *IEEE Fourth International Conference on eScience*, pages 253–260, 2008.

75. P. Marshall, K. Keahey, and T. Freeman. Elastic site: Using clouds to elastically extend site resources. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 43–52, 2010. doi: 10.1109/CCGRID.2010.80.

76. P. Marshall, H. Tufo, and K. Keahey. Provisioning Policies for Elastic Computing Environments. In *Proceedings of the 9th High-Performance Grid and Cloud Computing Workshop*, May 2012.

77. T. Matthey and et al. Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Transactions on Mathematical Software*, 30:237–265, September 2004. ISSN 0098-3500.

78. T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Gener. Comput. Syst.*, 25(5):541–551, May 2009. ISSN 0167-739X. doi: 10.1016/j.future.2008.06.013.

79. A. Merzky, K. Stamou, and S. Jha. Application Level Interoperability between Clouds and Grids. In *Grid and Pervasive Computing Conference, 2009. GPC &#039;09. Workshops at the*, pages 143–150. IEEE, May 2009. ISBN 978-1-4244-4372-7. doi: 10.1109/gpc.2009.17.

80. Microsoft Corporation. Microsoft windows azure platform. http://www.microsoft.com/windowsazure.

81. C. Moretti. Abstractions for Scientific Computing on Campus Grids, 2010.

82. C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain. All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids. *IEEE Transactions on Parallel and Distributed Systems*, 21(1):33–46, 2010.

83. C. Moretti, A. Thrasher, L. Yu, M. Olson, S. Emrich, and D. Thain. A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids. *IEEE Transactions on Parallel and Distributed Systems*, 23(12), 2012.

84. D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, page 9, Berkeley, CA, USA, 2011. USENIX Association.

85. A. Nicholls, K. A. Sharp, and B. Honig. Protein folding and association: Insights from the interfacial and thermodynamic properties of hydrocarbons. *Proteins: Structure, Function, and Bioinformatics*, 11(4):281–296, 1991. ISSN 1097-0134. doi: 10.1002/prot.340110407.

86. D. Nicol and J. Saltz. An analysis of scatter decomposition. *Computers, IEEE Transactions on*, 39(11):1337–1345, Nov 1990. ISSN 0018-9340. doi: 10.1109/12.61043.

87. T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurr. Comput. : Pract. Exper.*, 18(10): 1067–1100, Aug. 2006. ISSN 1532-0626. doi: 10.1002/cpe.v18:10.

88. A.-M. Oprescu and T. Kielmann. Bag-of-Tasks Scheduling under Budget Constraints. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 351–359. IEEE, Nov. 2010. ISBN 978-1-4244-9405-7. doi: 10.1109/cloudcom.2010.32.

89. S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 115–131, 2010.

90. P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. *ACM SIGOPS Operating Systems Review*, 41(3):289–302, 2007.

91. R. Pordes and et al. The open science grid. *Journal of Physics: Conference Series*, 78, 2007.

92. D. Rajan, A. Canino, J. A. Izaguirre, and D. Thain. Converting a High Performance Application to an Elastic Cloud Application. In *The 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2011)*, 2011.

93. D. Rajan, A. Thrasher, B. Abdul-Wahid, J. A. Izaguirre, S. Emrich, and D. Thain. Case Studies in Designing Elastic Applications. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

94. M. Resch, H. Berger, and T. Bönisch. A comparison of mpi performance on different mpps. In *Proceedings of the 4th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 25–32. Springer-Verlag, 1997.

95. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978. ISSN 0362-5915. doi: 10.1145/320251.320260. URL `http://doi.acm.org/10.1145/320251.320260`.

96. S. Sakr and A. Liu. Sla-based and consumer-centric dynamic provisioning for cloud databases. In *Proceedings of the IEEE Fifth International Conference on Cloud Computing*, pages 360–367, 2012. ISBN 978-0-7695-4755-8. doi: 10.1109/CLOUD.2012.11.

97. J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

98. T. Sandholm, J. A. Ortiz, J. Odeberg, and K. Lai. Market-Based Resource Allocation using Price Prediction in a High Performance Computing Grid for Scientific Applications. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 132–143. IEEE, 2006. ISBN 1-4244-0307-3. doi: 10.1109/hpdc.2006.1652144.

99. P. Sempolinski, D. Wei, D. Thain, and A. Kareem. A System for Management of Computational Fluid Dynamics Simulations for Civil Engineering. In *8th IEEE International Conference on eScience*, pages 1–8, 2012.

100. G. Shao, F. Berman, and R. Wolski. Master/slave computing on the Grid. In *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 3–16. IEEE, 2000. ISBN 0-7695-0556-2. doi: 10.1109/hcw.2000.843728.

101. Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038921.

102. M. Shirts and V. S. Pande. Screen Savers of the World Unite! *Science*, 290 (5498):1903–1904, Dec. 2000. ISSN 1095-9203. doi: 10.1126/science.290.5498.1903.

103. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

104. L. Stein. Genome annotation: from sequence to biology. *Nature Reviews Genetics*, (7):493503, 2001. doi: 10.1038/35080529. URL `http://www.nature.com/nrg/journal/v2/n7/full/nrg0701_493a.html`.

105. C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 71–84, Berkeley, CA, USA, 2005. USENIX Association.

106. H. S. Stone. High-performance computer architecture. 1987.

107. Y. Sugita and Y. Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chemical Physics Letters*, 314(1-2):141 – 151, 1999.

108. V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, Nov. 1990. ISSN 1040-3108. doi: 10.1002/cpe.4330020404. URL `http://dx.doi.org/10.1002/cpe.4330020404`.

109. M. Tanaka and O. Tatebe. Workflow Scheduling to Minimize Data Movement Using Multi-constraint Graph Partitioning. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 65–72. IEEE, May 2012. ISBN 978-1-4673-1395-7. doi: 10.1109/ccgrid.2012.134.

110. D. Thain. Coordinating Access to Computation and Data in Distributed Systems, 2004.

111. D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

112. A. Thrasher, Z. Musgrave, D. Thain, and S. Emrich. Shifting the Bioinformatics Computing Paradigm: A Case Study in Parallelizing Genome Annotation Using Maker and Work Queue. In *IEEE International Conference on Computational Advances in Bio and Medical Sciences*, 2012.

113. A. Thrasher, D. Thain, S. Emrich, and Z. Musgrave. Shifting the bioinformatics computing paradigm: A case study in parallelizing genome annotation using maker and work queue. *Computational Advances in Bio and Medical Sciences, IEEE International Conference on*, 0:1–6, 2012. doi: http://doi.ieeecomputersociety.org/10.1109/ICCABS.2012.6182647.

114. B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39, Mar. 2008. ISSN 1556-4665. doi: 10.1145/1342171.1342172.

115. B. Vandalore, W. chi Feng, R. Jain, and S. Fahmy. A survey of application layer techniques for adaptive streaming of multimedia. *Real-Time Imaging*, 7(3):221 – 235, 2001. ISSN 1077-2014. doi: http://dx.doi.org/10.1006/

rtim.2001.0224. URL http://www.sciencedirect.com/science/article/pii/S1077201401902244.

116. C. Vecchiola, S. Pandey, and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 4–16. IEEE, Dec. 2009. ISBN 978-1-4244-5403-7. doi: 10.1109/i-span.2009.150.

117. D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Transactions on Internet Technology*, 7 (1), Feb. 2007. ISSN 1533-5399. doi: 10.1145/1189740.1189747.

118. R. Weber, A. Gothandaraman, R. Hinde, and G. Peterson. Comparing hardware accelerators in scientific applications: A case study. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):58–68, Jan 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2010.125.

119. M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, 2001. ISBN 1-58113-389-8. doi: 10.1145/502034.502057.

120. A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, page 27, 2012.

121. M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11):50–60, Nov 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.365.

122. R. Wolski, J. Brevik, G. Obertelli, N. Spring, and A. Su. Writing programs that run EveryWare on the Computational Grid. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1066–1080, Oct. 2001. ISSN 1045-9219. doi: 10.1109/71.963418.

123. L. Wu, S. Garg, and R. Buyya. Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 11th IEEE/ACM International Symposium on*, pages 195–204, 2011. doi: 10.1109/CCGrid.2011.51.

124. L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, 2008.

125. J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, Sept. 2005. ISSN 0163-5808. doi: 10.1145/1084805.1084814. URL http://doi.acm.org/10.1145/1084805.1084814.

126. L. Yu. Right-sizing Resource Allocations for Scientific Applications in Clusters, Grids, and Clouds, 2013.

127. L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain. Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions. *Journal of Cluster Computing*, 13(3):243–256, 2010.

128. M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

129. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, page 10, Berkeley, CA, USA, 2010. USENIX Association.

130. M. J. Zaki and C.-T. Ho. *Large-scale parallel data mining*. Number 1759. Springer, 2000.

131. Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the Fourth International Conference on Autonomic Computing*, pages 27–, 2007. ISBN 0-7695-2779-5. doi: 10.1109/ICAC.2007.1.

132. Q. Zhu and G. Agrawal. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. *IEEE Transactions on Services Computing*, 2012. ISSN 1939-1374. doi: 10.1109/tsc.2011.61.

This document was prepared & typeset with LaTeX 2_ε, and formatted with ND**diss**2_ε *classfile (v3.2013[2013/04/16]) provided by Sameer Vijay and updated by Megan Patnott.*