Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Signature:

Jaroslaw K. Slawinski	Date

Adaptive Approaches to Utility Computing for Scientific Applications

By

Jaroslaw K. Slawinski
Doctor of Philosophy

Computer Science

Vaidy Sunderam, Ph.D.
Advisor

Ken Mandelberg, Ph.D.
Committee Member

Alessandro Veneziani, Ph.D.
Committee Member

Accepted:

Lisa A. Tedesco, Ph.D.
Dean of the Graduate School

Date

Adaptive Approaches to Utility Computing for Scientific Applications

By

Jaroslaw K. Slawinski
Ph.D., Emory University, 2014

Advisor: Vaidy Sunderam, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the Graduate School
of Emory University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in Computer Science
2014

Abstract

Adaptive Approaches to Utility Computing for Scientific Applications
By Jaroslaw K. Slawinski

Coupling scientific applications to heterogeneous computational targets requires specialized expertise and enormous manual effort. To simplify the deployment process, we propose a novel adaptive approach that helps execute unmodified applications on raw computational resources. Our method is based on situation-specific "adapter" middleware that builds up target capabilities to fulfill application requirements, avoiding homogenization that may conceal platform-specific features. We investigate three dimensions of adaptation: performance, execution paradigm, and software deployment and propose the ADAPT framework as a methodology and a toolkit that automates execution-related tasks. For parallel applications, ADAPT matches logical communication patterns to physical interconnect topology and improves execution performance by reducing use of long-distance connections. In a proof-of-concept demonstration of application–platform paradigm transformation, ADAPT enables execution of unmodified MPI applications on the Map–Reduce Platform as a Service cloud by recreating and emulating missing MPI capabilities. To facilitate software deployment, ADAPT automatically provisions resources by applying soft-install adapters that dynamically transform target capabilities to satisfy application requirements. As a result of these types of transformations, a broader spectrum of resources can smoothly execute scientific applications, which brings the notion of utility computing closer to reality.

Adaptive Approaches to Utility Computing for Scientific Applications

By

Jaroslaw K. Slawinski
Ph.D., Emory University, 2014

Advisor: Vaidy Sunderam, Ph.D.

A dissertation submitted to the Faculty of the Graduate School
of Emory University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in Computer Science
2014

Acknowledgments

Research supported in part by US National Science Foundation Grant OCI-1124418.

The author thanks collaborators Dr. Tiziano Passerini, Dr. Umberto Villa, and Prof. Alessandro Veneziani.

I would like to express my gratitude to my advisor, Prof. Vaidy Sunderam, for his support, patience, and encouragement throughout my graduate studies.

To my Wife

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

*aaS Everything as a Service

*nix Unix-like operating systems

ADAPT Adaptive Application and Platform Translation

ADAPT-D ADAPT Deployment

CFD Computational Fluid Dynamic

DLL Dynamic-link library

FEM Finite Element Method

HPC High-performance computing

IaaS Infrastructure as a Service

MPI Message Passing Interface

MPIMR MPI in Map–Reduce

MR Map–Reduce

NSE Navier–Stokes equations

PaaS Platform as a Service

PDE Partial Differential Equation

QoS Quality of service

RDE Reaction–Diffusion equation

SaaS Software as a Service

SaE Scientific and engineering

SSH Secure Shell

VO Virtual organization

Chapter 1

Introduction

Cyber-infrastructure is enabling significant advances in every domain. With the increasing maturation of grids and especially clouds, the vision of computing as a utility, is starting to become a reality. However, the usability of cyber-infrastructure platforms for efficiently executing science and engineering applications is proving to be a challenge. Such applications are often able to utilize resources that span local and campus facilities, those accessed via virtual organizations, and on-demand Cloud offerings. In order to do so, they often need target-specific adjustments and reconciliations which pose considerable logistical obstacles to effectively and flexibly permitting the use of the best resource in a given instance.

These problems become of immediate import as in recent years resource pools have expanded from local clusters and servers to encompass grids and other types of sharing enabled by VOs, and recently to public and private Cloud platforms. Also, progress in technology makes powerful workstations and affordable many-core clusters that can handle SaE-class applications affordable for the users. In this context, on-demand access to resources, without the burdens of ownership, maintenance, or even operating expertise is very appealing and more users are exploring these innovative HPC platforms. However, this usage shift is not without its drawbacks. When cloud computing transformed "exclusive" HPC into "affordable" HPC and made SaE computing available to many users, *usability* is often severely hampered, particularly for legacy applications. The major obstacle is *heterogeneity* in local, grid, and cloud computing platforms—not only in terms of machine architectures and network interconnect but in execution performance, software deployment as well as programming and execution paradigms. This variability poses serious challenges in

the ability to select or switch between providers for availability and cost reasons. For example, executing the same application for different runs on an on-premises cluster, a remote Teragrid resource, and a collection of Amazon EC2 [98] instances is a logistically complex and cumbersome process.

1.1 Goals

1.1.1 Executability Enhancement

This dissertation investigates novel approaches to enhance the executability of SaE applications on varied computational back-ends including different types of clouds as well as grids and on-premise resources without necessitating program changes or manual effort. We believe that at least for certain categories of applications and for most target platforms, defining a unified capability interface, and subsequent application-to-platform mapping is possible, and that such flexibility will be of substantial value. We envision that this research helps evolve cross-usage of on-premises, grid, and cloud computing platforms, identify opportunities for unification, and reduce, or even eliminate, otherwise inevitable porting efforts. This, in turn, will contribute to increased adoption of resource sharing in more domains, enable users to focus on their applications rather than on unproductive conversion logistics, and move a step further towards realization of "Computing as a Utility."

1.1.2 Performance Adaptation

One of the exploratory goals is to check if execution of parallel applications can be automatically adapted to the actual computational target in order to deliver the best possible execution performance. This topic is crucial for SaE applications as suboptimal execution severely influences scalability which, in turn, may exclude a particular platform from being used. In this respect, automatic performance adaptation increases the set computational resources available for the user.

1.1.3 Deployment Adaptation

Another goal is to facilitate deployment of SaE applications on a wider range of computational resources. SaE software conditioning is particularly difficult as these applications are usually

distributed in the form of source codes, require multifarious, nontrivial, and numerous dependencies as well as utilize parallel and distributed programming paradigms. Similarly, SaE software dependencies alone are deployment-hard and often require atypical installation methods. We want to enhance usability of SaE applications beyond on-premise or supercomputer center machines and offer the SaE software on any architectures accessible for the user, including department clusters, grids, or IaaS clouds. We aim to embrace the heterogeneity resulting from using a variety of targets by building applications from sources. We expect that this helps extricate users from the burden related to an unproductive software deployment phase and promote switching between targets and vendors for availability or financial reasons, even for a single run of an application.

1.1.4 Paradigm Adaptation

The final goal is to investigate the complementary diversity in application programming and execution paradigms with a view to understanding their commonalities. Many models are used in SaE applications, typical among which are: (1) Parallel programs (typically MPI-based), (2) Script-based, loosely defined to include interactive and batch processing using packages such as MATLAB, and (3) Workflows, characterized by interacting sub-applications, each of which may use a different model and execute on different types of platforms. Paradigms such as MapReduce [1], parameter sweeps [2], or Global Address Space programming models [3] are assumed to be included in one or more of these broad classes. Users develop SaE applications using these models to execute, either by default or by design, on a specific target platform. When multiple types of resources are available, they are constrained in the ability to execute their applications on unfamiliar targets, thereby sacrificing effectiveness and possibly efficiency. However, if functionality equivalent to the specific needs of an application could be obtained through transformed expression of their requirements, they could run on target platforms other than those for which they were originally designed.

1.2 Contributions

This section signals the main research outcomes. The detailed contribution descriptions are provided in chapters that describe the concepts, performed experiments, and proof-of-concepts deliverables.

1.2.1 Executability Enhancement

In this dissertation, we report our research effort on the *Adaptive Application and Platform Translation* (ADAPT) framework that aims at matching applications to specific resource back-ends thereby *increasing flexibility* in users' choices with respect to target platforms, *runtime transformations* to address performance tuning important for SaE applications, and *conditioning environments* to automatically prepare back-ends for execution. The key concept is that *unmodified* applications can be executed on multiple types of computational platforms with the assistance of an application–target-specific middleware that is dynamically and automatically assembled from *software adapters*. Our idea is to enhance, or specialize, platforms via environment or runtime software conditionings to enable this flexible use of multifaceted resources. We verified the adapter-based design in a few proof-of-concept experiments and provided further insight showing how this approach helps to make the utility-computing model feasible for existing applications.

1.2.2 Performance Adaptation

In order to understand factors that influence performance of SaE parallel applications, we benchmarked a set of parallel heterogeneous computational architectures using an HPC-class hemodynamic application. We learned and confirmed that computational fluid flow simulations for highly unstructuralized objects, such blood vessels, are sensitive to communication imbalance and proper mapping of parallel processing elements into physical network topology must be done to achieve acceptable execution performance. We showed that the importance of the "right" placement strategy increases with communication heterogeneity in the target chosen for execution. As a result, using machines different than typical for science such as IaaS clouds or ad-hoc peer-to-peer assemblies requires extra attention; otherwise, these platforms must be avoided as they cannot compete with HPC platforms in the time-to-completion and operational cost attributes. In a proof-of-concept example, we use graph mapping techniques to balance communication requirements and prove that information needed to perform this matching may be obtained directly from the input data without resorting to benchmarks.

In another set of tests, we studied trade-offs between performance (time-to-completion metric) and cost of heterogeneous platforms. We modeled three utility use cases and verified them against

the benchmark results. We showed that in some situation using cloud resources is justified for SaE applications even if they are generally slower than specialized computing targets. Especially in time critical situations, using on-demand machines that are available immediately may surpass benefits from using a supercomputer with hours-long job queue. Overall, performance adaptation is an important aspect of utility computing as it limits the factual costs of execution and broadens the spectrum of resources that can be considered for application execution.

1.2.3 Deployment Adaptation

We studied common issues in SaE software deployment, understood sources of difficulties, and devised an improved model of SaE software deployment that is contrary to current approaches. Our idea is to induce deployment errors and treat them as feedback from the runtime. This dynamic application–target-specific information steers the deployment steps that dynamically apply environment conditioning adapters (deployment recipes). Moreover, our method facilitates not only software deployment but also easies execution itself. As the scientific contribution, we propose new execution and deployment algorithms and outline deployment model.

In addition, we implement the automatic and dynamic multi-target deployment toolkit `ADAPT-D` that integrates currently separate deployment phases for an application and its dependencies. This software (1) captures diverse users' activities leading to installation of software components on a given platform and (2) processes deployment knowledge for reusing in other deployment contexts. Adoption of `ADAPT` should help developers and end-users as it reduces deployment maintenance for both groups. The delivered deployment toolkit can be used for a wide spectrum of machines, from typical SaE targets, such as supercomputers and high-end clusters, to single workstations and virtualized platforms, such as grids and IaaS clouds. Such the improved software provisioning (1) enhances usability of heterogeneous machines as they become instantly ready to run users applications as well as (2) improves productivity at HPC centers as the software deployment can be automated.

1.2.4 Paradigm Adaptation

In order to explore the application–execution paradigm matching and fully confirm usability of ADAPT propositions, we explored possibilities to execute parallel MPI applications on Map–Reduce Platform as a Service clouds. As the result of these studies, we developed the MPI-MRE software product that provides the MPI libraries and typical command line interface for Hadoop-based platforms. Thanks to provided software adapters, the users can execute unmodified MPI applications that shun point-to-point communication operations (only collective operations are permitted) on Map–Reduce infrastructures. In this investigation, we entirely rebuild communication capability—indispensable for MPI applications—using distributed file system capability provided in Hadoop. This shows that the concept proposed in ADAPT, i.e., using adaptable middleware to reconciliate application requirements with target capabilities, is feasible. Specifically for our example, execution of MPI applications in Map–Reduce can be also viable as the created execution environment transparently acquires Map–Reduce-specific features, e.g., the fault tolerant execution. Even if some types of application–target coupling may seem to be "academic" or outlandish, this increases understanding of commonalities between application and execution paradigms and may lead to new paradigms that support both modern and legacy applications.

1.3 Outline

Chapter 2 describes the ADAPT ideas; some research extensions and proposition of another types of adaptations are included in [4]. In Chapter 3 we report experiences with deploying and executing an SaE application on different types of computational platforms. This study, first reported in [5], provides invaluable insight for performance and deployment considerations. In the next chapters 4 and 5, we deeper study these issues; more information about these topics is included in [6, 7, 8]. In Chapter 6 we describe deployment adaptation and automatic deployment toolkit. Preliminary work on this topics can be found in [99, 9, 10, 11]. Exploration of paradigm adaptation experiences are given in Chapter 7; more comprehensive reports are in [12, 13]. Finally, Chapter 8 concludes this dissertation and gives another research directions.

Chapter 2

Research Overview

The key concept of this research is that several classes of *unmodified* applications can be executed on multiple types of computational back-ends with the assistance of a flexible and adaptive middleware environment. Figure 2.1 illustrates the conceptual goals of `ADAPT`, showing applications using (1) multiple resources and resource classes for a single run and (2) different types of resources for different executions. The `ADAPT` framework dynamically provides *matching adapters* and performs target platform *environment conditioning* as needed to enable this flexible use of multifaceted resources. For example, MPI applications such as Gromacs [14] or NPB [15] typically run on clusters managed by a batch scheduler, but can be also executed on workstation networks or IaaS clouds [16, 17]. In the latter situations, `ADAPT` assists with the required provisioning and staging needed to prepare the target environment but may also supply simple adapters, e.g., command-name replacements to emulate typical MPI or job scheduler operations.

Moving to somewhat more sophisticated scenarios, script-based applications written assuming a certain execution environment (specific libraries, data access methods, OS versions) can be adapted for other targets through the use of wrappers, software packages offering equivalent functionality, and other similar transformations. "Adapters" to enable such matching can be assembled from repositories or even dynamically generated and deployed automatically, thereby facilitating execution on a compatible target platform.

In an even more complex scenarios, it is possible to imagine paradigm transformation, e.g., executing an iterative MapReduce application as an interacting set of MPI processes [18]. At the extreme end of the scale, the entire (sub)functionality of an application can be realized by

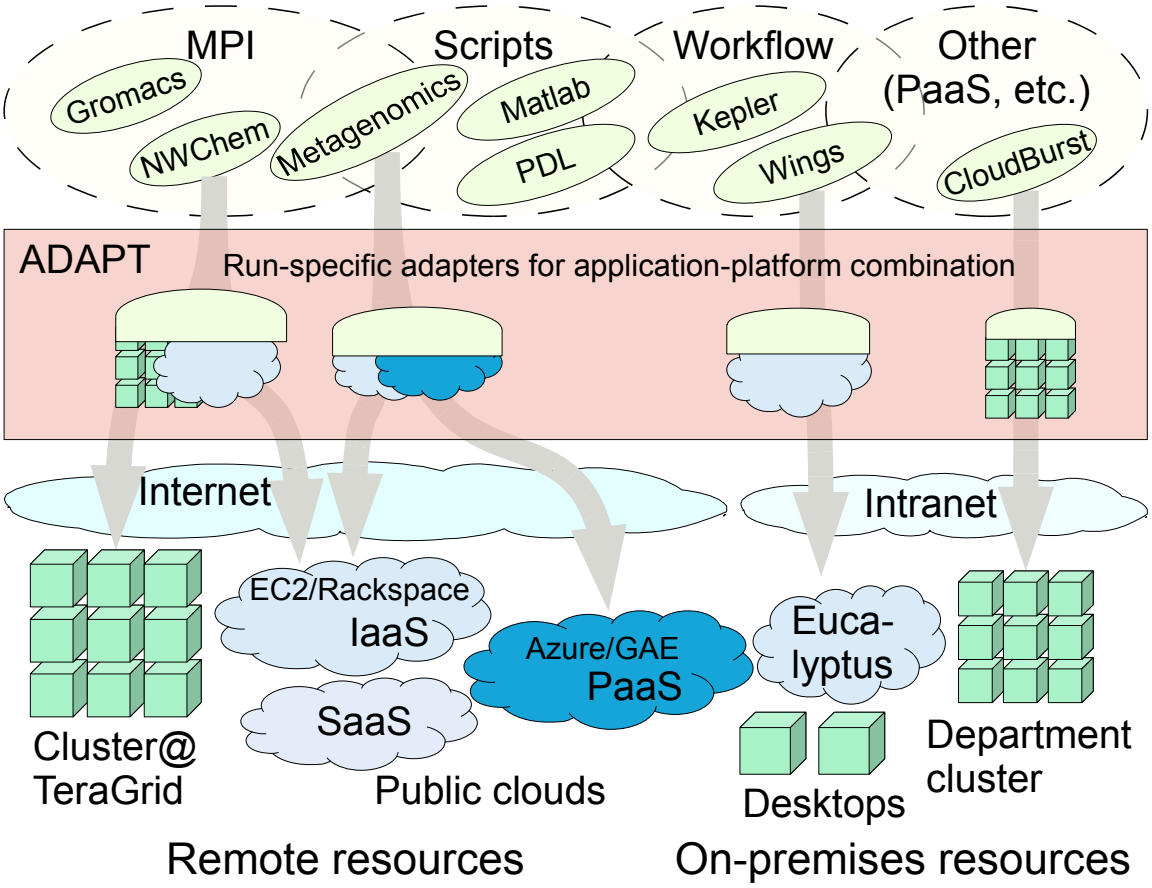


Figure 2.1: ADAPT conceptual overview

an equivalent "outsourced" substitute that executes on an available (or best suited) platform, or is simply delivered via a SaaS cloud. An illustration of the range and level of such adaptations is shown in Figure 2.2. The ADAPT investigates abstractions for representing and matching application needs to resource capabilities and evolve a suite of adapters to enable cross-platform execution.

The goal of the ADAPT project is pragmatic: it (1) aims to enable application execution without excluding resource types to the maximum extent possible and (2) relieves both users and resource providers from burdens of application porting[1] as well as resource provisioning and coordination. To achieve this ADAPT intends to offer complementary mechanisms for (1) provisioning of system software on the resource side and (2) mapping of program needs on the application side, as shown in Figure 2.1 and 2.2. This two-pronged approach has the potential to improve executability of unmodified applications on raw (i.e., unconditioned) resources, even if the resource presents a

[1] We mean "porting" not in the conventional program-level sense but rather in the deployment sense, i.e. libraries, environment, file requirements, monitoring facilities, etc.

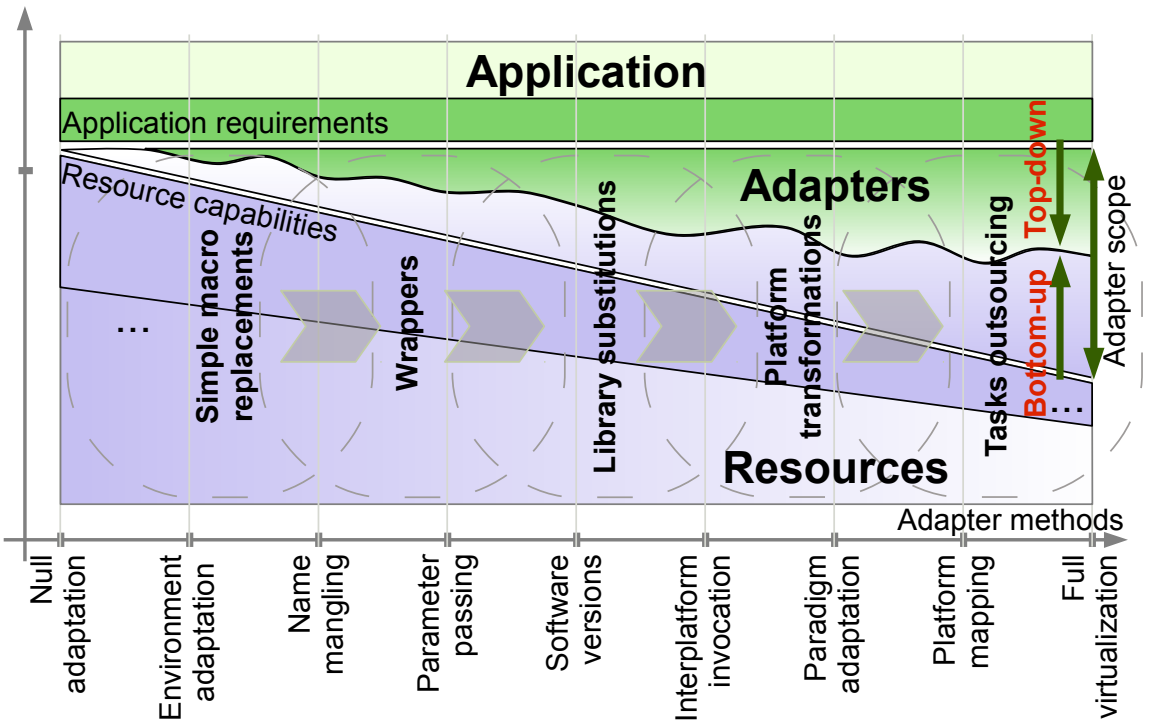


Figure 2.2: Adaptation must address differences between the persistent application requirements and changing target capabilities. Only few platforms may instantly execute the application and most resources will require adaptation of a different scope.

non-obvious, non-standard execution back-end for this application. To achieve this, ADAPT will support *dynamic* and *automatic preconditioning* of resources together with provisioning *adapters* for applications where possible. Examples for the former include automatic meeting software dependencies such as libraries or run-time systems needed to execute an application on user-selected computational back-ends. Stubs or macros for parameter matching, library interceptors [19], and callback based translations are examples of the latter.

To be more concrete, as our targeted domain is an SaE applications class, we intend to focus more on message passing parallel paradigm, exemplified by MPI, that is the main programming scheme in scientific applications. MPI applications require process management facilities, an efficient communication fabric, and a responsive I/O subsystem to attain scalability with good execution performance [20, 21, 22]. Although not every target resource can present such characteristics, many resources are able to sustain MPI execution and may offer valuable execution environments for testing or solving smaller problems. ADAPT intends to facilitate execution of such applications by conditioning target environments automatically, i.e., by staging the required libraries

(e.g., MPI, numerical), ensuring version compatibility, and providing mechanisms for relaying process status, monitoring, and trace data.

In terms of platform types, our goal is to address seamless execution on as many resources (and combined aggregations thereof) as possible. We view resource types as characterized along multiple dimensions. Access methods range from the interactive (SSH), to batch and queue systems (e.g., Condor, Torque), to specialized portals (e.g., Amazon Elastic MapReduce [100], scientific gateways [23]). Functionally, process management capabilities may vary significantly from simple start/stop interfaces to extended support of executed tasks. Differences are also prevalent on such levels as software capability discovery (some resources may provide metainformation), allocation management, and application staging. The key idea of `ADAPT` is to virtualize these capabilities and provide abstractions of services such as execution, staging, process control, monitoring, and tracing. However, we refrain from attempts to homogenize or standardize interfaces to avoid concealing valuable capabilities or forcing users to modify their applications. In addition to traditional scientific computing platforms, we may target on-demand resources—specifically IaaS, PaaS, and even SaaS clouds. From a high-level, utility perspective, capabilities of such resources may be abstracted in their native forms but also as equivalent to other resources by *conditioning* them via appropriate chunk allocations, image selections or site-preparation, and soft installation of software subsystems [101]. Doing so may have advantages from an economic and availability standpoint but can be cumbersome—`ADAPT` aims to transparently and dynamically handle the preparatory steps in *aaS resource usage making them valuable execution infrastructures.

2.1 Research Issues

At a very high level, our major research focus is to enable executability of legacy and evolving applications on current and emerging target platforms. Contrasted with approaches that attempt to standardize programming models or languages, we adopt a systems-oriented philosophy of provisioning adaptive middleware that enable applications and resource platforms to be matched to each other. We focus on the following research challenges: (1) specifying, matching, and mapping application requirements with resource capabilities and (2) assembling adapters and environment conditioning.

Determination of an application's computing, storage, and interaction requirements is the obvious prerequisite to enabling its execution on a given target platform. In the ADAPT project, we interpret "requirements" to primarily imply functionality—partly in terms of the programming model but mostly in terms of infrastructural and environmental setup. As mentioned earlier, we intend to focus on a major class of SaE applications, i.e., MPI programs. The core requirement of that applications is an MPI-capable environment. However, many applications require more than simply a standard version of a compatible MPI library. Often, extensions such as those for monitoring, checkpointing, or profiling are assumed. Similarly, beta-features (e.g., one-sided communication) or particular runtime systems (e.g., tunneling across clusters) might be hardcoded into applications. Input and output performed by the application may assume a certain configuration of process rank placement relative to file location. A major challenge for ADAPT is to be able to evaluate to the extent possible, a complete set of needs that an executing application is predicted to require. We note briefly here that our approach to this issue will include concise and pragmatic descriptors, assessment during preparatory stages of execution, and simple analysis of runtime parameters.

Complementary to the identification of application requirements is the description of resource capabilities. The research challenges in this respect pertain to specification of functionality, capacity, and performance. More importantly, from the perspective of this research and wider usability of cyber-infrastructure, determination of equivalence—either directly or through transformation—is crucial. For example, consider a resource represented by a shell access point (e.g., SSHD) that supports process spawning, file creation and deletion. An equivalent capability can be readily imagined on an IaaS instance, subsequent to a series of staging steps. Now consider a parallel cluster capable of accepting batch queuing requests for parallel MPI programs. A similar capability can be assembled as a collection of remote shells or IaaS instances, with appropriate preparatory and aggregation conditioning. Effective and practical schemes to dynamically assess and project such *unified*, *composable* resource capabilities is the focus of our efforts in this regard.

Virtualizing the capacity and performance dimensions also poses interesting questions and is of considerable importance in this project given its goal of facilitating production applications. It is envisioned that some types of "universal" attributes can be characterized and quantified in their virtual capability representations but others such as scheduled or dynamic resources may prove

more difficult. We focus primarily on capacity and performance metrics that are common across multiple platforms for the same type of resource (computation, storage, interaction) and which can be expressed in meaningful quantitative terms. It is also interesting to include resource descriptions, metadata and dynamic probes to enable rich characterization of resource functionality as well as service levels.

Evolving a methodology for the assembly of adapters is a major research issue. Once application needs and resource capabilities are identified, the subsequent challenge is to determine how to match or, where possible, reconcile these needs to the given target environment. As previously discussed, certain types of adapters are likely to be straightforward to generate; these include name substitution, parameter mapping, and similar syntax-oriented transformations. Even in these instances, schemes to incorporate and deploy adapters will necessitate the investigation of different strategies, e.g., macro processing vs. runtime translators. For more complex adapters, adapters may consist of substitute libraries. As an example, consider an application that is written in terms of a beta-feature, e.g., MPI one-sided communication (`MPI_Put`, `MPI_Get`). If the target platform does not support the appropriate version of the MPI library, an adapter implemented as a series of standard MPI-1.1 calls can accomplish the necessary matching [24]. Such adapters may either be built into the `ADAPT` framework, retrieved from a library, or provided through interaction with the user.

More involved situations could necessitate runtime substitution of calls to unavailable (e.g., numerical) libraries by equivalent calls to third-party SaaS services. Evolving a systematic framework to accomplish such analyses, identifying and including the adapters to implement them, and coordination with the user/application are the research challenges that we will address in this context. Another intriguing notion that deserves in-depth investigation concerns composability of adapters. Dynamically assembling an adapter stack from previously created components will be of considerable value but is also challenging to design and implement.

Before the application can be launched, the selected target environment should be conditioned to ensure that dependencies are satisfied. Aspects of this phase include staging of the appropriate libraries, input and output files, and ensuring that environment variables are set correctly. Mechanisms to accomplish this can leverage tools such as GNU Make and Autoconf. Nevertheless, dynamic dependency resolution remains a significant and unaddressed research issue in resource sharing environments. Since grids tend to couple resource allocation with job submission, users

are currently forced to prepare stageable bundles in advance. On IaaS cloud instances, the selected image may not satisfy all of the application's dependencies. New mechanisms to detect and address such issues will be explored in the course of our work on target environment conditioning.

2.2 Adapters

Adaptation may be required on many levels and for many reasons. The most important type of adaptation is the one that enables execution. In another situation, the users may request a parallel execution platform assembled from peer-to-peer or IaaS cloud nodes which may need multistage and multilevel preparation. In yet another scenario, the users may want a specific execution interface to available resources, e.g., a PBS job scheduler-like interface spanning a few workstations. A simple example is shown in Figure 2.3. The application [25] requires a specific library that is unavailable on the target resource. `ADAPT` retrieves a "shim" library that maps the required calls to equivalent functions available on the target, and stages the shim library in the correct location with the correct name and library path. Such shims may either be built-in to `ADAPT` for commonly required mappings or be obtained from a repository collection of contributed adapters.

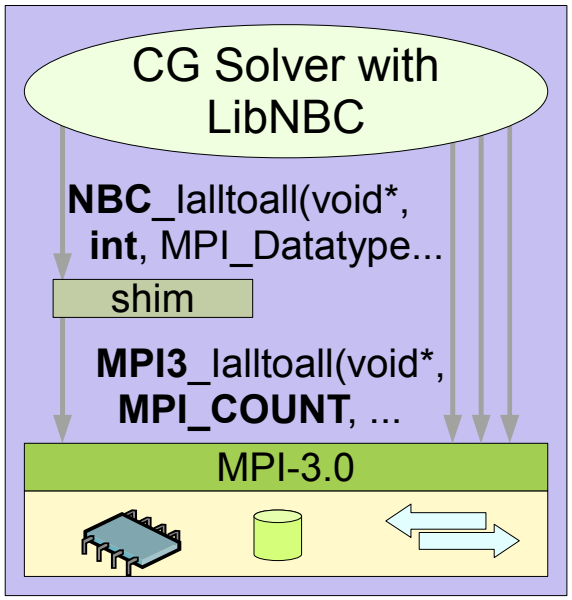


Figure 2.3: Providing library substitution

Adapters ensure that execution needs are met on a particular target for a specific application. Adapters therefore serve as a *situation-specific middleware layer* between (cf. 2.2). Depending on

the circumstance, adapters may be sequences of shell commands or library shims interfacing to equivalent functionality. Other variations of adapters are also possible. Adapters may be modeled to fit the dependency–capability gap between the application program requirements and the target resource capabilities, as it is shown in Figure 2.4.

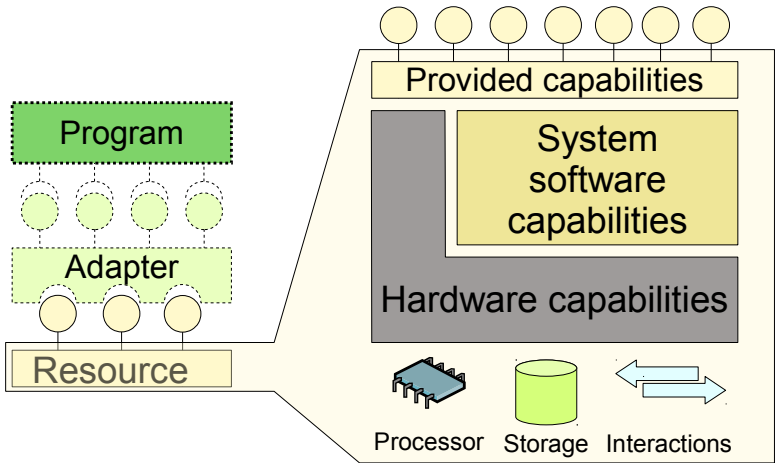


Figure 2.4: A software adapter is needed to match specific application requirements with given target capabilities.

Figure 2.5 shows example possibilities for a single application that is intended for one resource target but may execute on others after adaptation. The native scenario shown on the left illustrates a "null" adapter, while on a very similar target platform, only minimal adaptation might be necessary. On a substantially different, but nonetheless compatible platform, substantial requirement-to-capability model may be needed. Finally, it may be possible to have advanced adapters, which at runtime accomplish application requirements by invoking an external "outsourced" service, as shown in Figure 2.6. Such callback operations may be used as a foundation for providing capabilities that a particular resource does not natively support (e.g., file operations for some PaaS clouds).

In addition, `ADAPT` may provide systematic approaches for complex adapters and composition of adapters. For instance, a specific application-target pairing may require adapters for program-library invocation issues (name mangling, parameter re-ordering), different library forms (static, shared, object libraries), and incompatibilities at the system level (Windows, *nix, portable virtual machines). To reconcile various technical constraints, we might develop a set of specialized versions of adapter elements, each for a respective aspect of the matching issue. Experimentation should

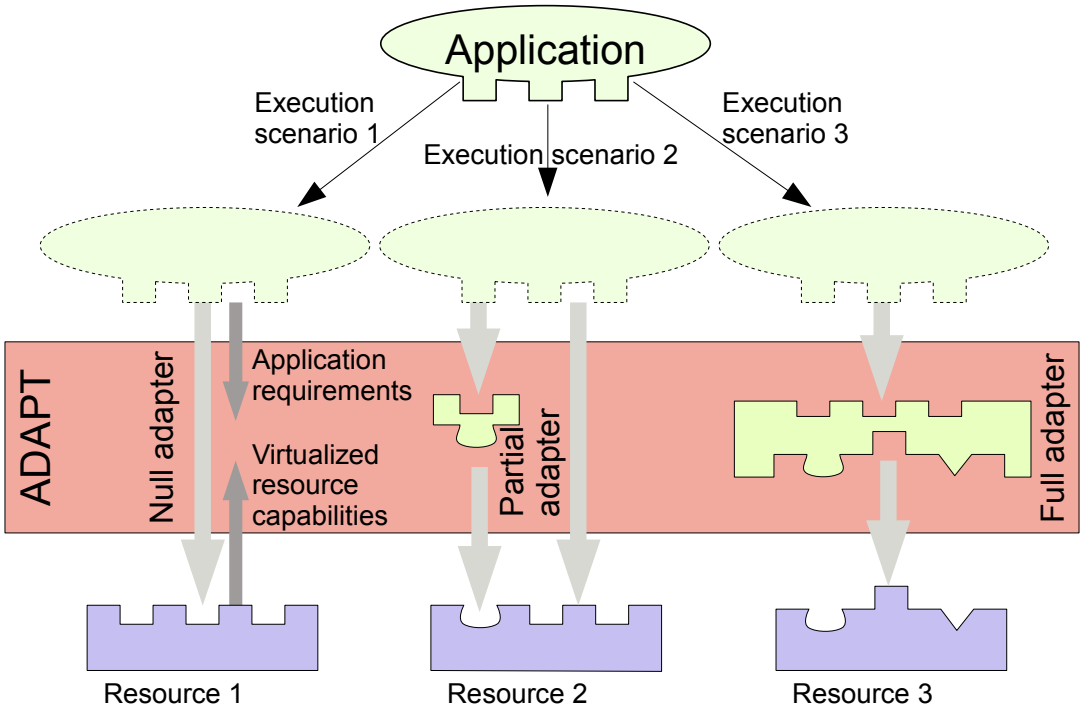


Figure 2.5: Situation-specific adapter inclusions

allow to estimate whether monolithic adapters created by combining appropriate elements are more effective that composing general purpose adapters and shims.

2.3 Applying Adapters to Programs

From a pragmatic point of view, there are a few options to investigate on how to apply library adapters to a program. Static linking may be the easiest method to "inject" dependencies into the program executables. Providing dynamic libraries may be possible by the DLL injection technique (`LD_PRELOAD`, `LD_LIBRARY_PATH`, `ld.so.conf`, `*.local`, etc.). It may also enable adaptation of an already compiled application as the requirements are to be resolved during execution. More sophisticated mechanisms for adapter delivery provide operating system level virtualizations and sandboxing, however not always these supreme techniques are available [19, 102, 26]. Finally, we may examine interception of system calls in order to capture external program invocations, e.g., UMview [27], ptrace [28], process monitors, or virtualization of file operations, e.g., FUSE [103]. The last technique may be the only possibility for applications available as binary packages.

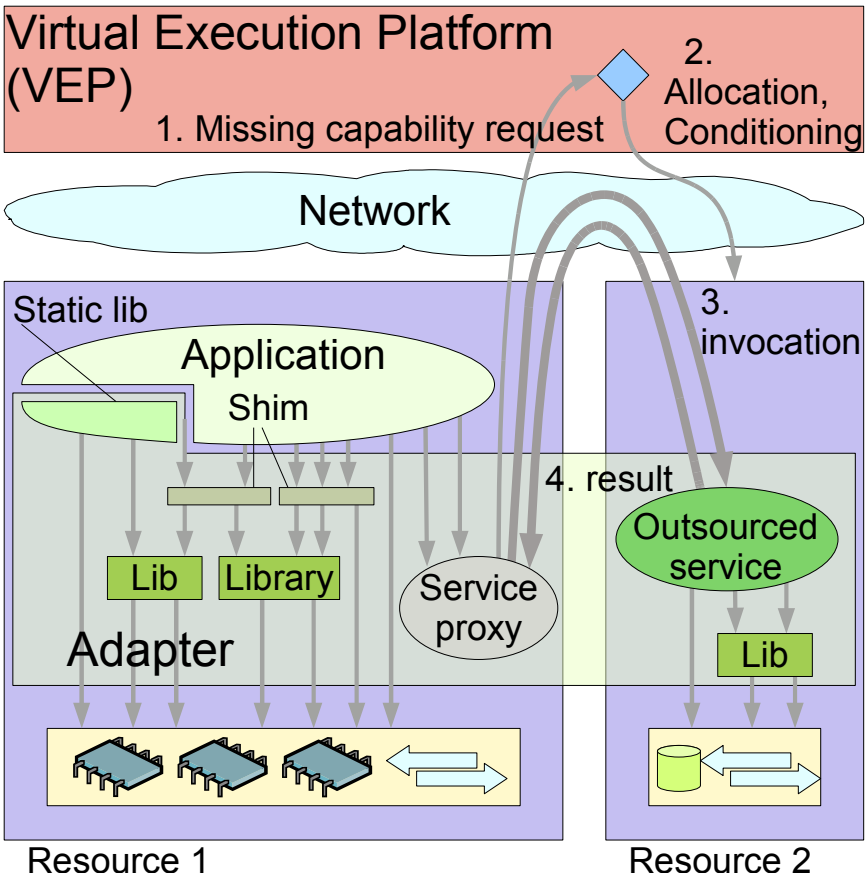


Figure 2.6: Outsourcing task execution

2.4 Related Work

The rapid growth and popularity of cloud computing and *aaS resources has increased the variety and heterogeneity of platforms available for SaE applications. Local resources, grids, and now Clouds offer a wide spectrum of capabilities, parameterized by capacity, availability, cost, and usability. The last attribute is especially important in realizing the vision of "computing as another utility" and is the focus of this proposal.

In order to enable portable application execution, one approach is to provide homogenization at the access level and application paradigm level. In the grid space, methods and techniques put forth by the Globus [29] project are canonical examples of standardization. In the cloud domain, there are several efforts aimed at access homogenization through formal and informal standardization efforts such as Simple Cloud API [104], Open Cloud Manifesto [105], EUCALYPTUS [30], Nimbus [31], and AppScale [32].

In order to support homogenization at the paradigm level, either (1) resource providers offer specialized cloud services, such as Amazon Elastic MapReduce [100], Tashi [33], or (2) users may adapt an resource to a required specialization level via conditioning, i.e., installing relevant middleware layers, e.g., ElasticWolf [16], Unibus [34], Nimbus [31]. There are also projects that, instead of the homogenization approach, propose new application frameworks that provide their own programming models (e.g., CometCloud [35] and Aneka [36]). Our current ADAPT project adopts a somewhat different philosophy and uses a blend of virtualization and middleware adapters to assist with cross-target execution. This approach permits specific features of resources to be fully exploited, while permitting sufficient flexibility in matching applications to the best suited resource for a given run.

The complementary aspect of facilitating application execution is to ensure that the selected target contains all the needed dependencies, libraries, and runtime systems. In IaaS clouds such as Amazon EC2 [98] or Rackspace [106] one could argue that virtualization addresses these issues, i.e., an image containing all the needs of the application could be constructed. However, this approach is not ideal when some requirements vary form run to run (e.g., the use of beta vs. production libraries) or when switching between providers. These drawbacks apply also to projects that statically link executables, creating application bundles with all needed libraries (PortableApps [107]), or those that prepare specialized images with preinstalled software dependencies (rPath, rBuild [37]). Furthermore, such approaches are inapplicable in shared resource or grid environments that have to be prepared manually. In this project we leverage our past and ongoing work on provisioning application requirements through environment conditioning and software-assisted staging of executables, libraries, and other dependencies. We believe that our research on a virtualized execution platform realized via middleware adapters and preparing target resources through conditioning will help reduce logistical and operational burdens on both providers and users. Additional reviews of related work and concepts are included in the following chapters that discuss in-depth selected aspects of `ADAPT`.

Chapter 3

Background and Context

Computing as a utility has become a reality in many domains; computational clouds deliver storage and processing resources on-demand via methods analogous to more traditional utilities. Such a paradigm is evolving for high-performance science and engineering applications (*High-Performance Computing*—HPC). Typically, applications in the HPC domain are characterized by computing and/or data intensive codes that are parallelized explicitly, commonly based on the message passing programming model. These applications largely execute on local, on-premise clusters or on platforms referred to as computational grids—although in practice, grid-computing predominantly manifests simply as remote access to clusters, just in a different administrative domain. In both settings, it has been traditional to measure the performance of HPC applications by a single metric viz. *time to completion* for the particular application in question, parameterized along two dimensions: problem size and number of processing elements used.

With the advent of cloud computing, two interesting perspectives have become relevant: (1) the viability of executing parallel applications on the cloud (either through self-assembly or renting a pre-built cluster) and (2) the actual dollar cost effectiveness of executing HPC applications on different target platforms. In this research experiment we report on experiences with executing a Finite Element Method (FEM) code on four different platforms that are heterogeneous in secondary respects (interconnect, access method, use cost) and attempt to characterize the overall "expense factor" of each. We provide some background information on normal modes of scientific application execution and subsequently outline the FEM code used in our exercise. We then describe the process and issues involved in *preparing* and *deploying* the application on four different platforms.

Measurements of execution time, augmented with usage cost and (qualitative) deployment effort are presented and discussed; the contribution concludes with a summary of factors that characterize the effectiveness of using different kinds of platforms. For the ADAPT project, it is a corner stone example of hard to deploy SaE-class software.

3.1 An Illustrative Example

HPC is intrinsic and integral to most fields of scientific endeavor. Message passing parallel programs are a staple modality of numerical simulations and computational analyses. In addition to the parallel framework, e.g. MPI, codes depend on various other auxiliary components: scientific and mathematical libraries, header files, or particular compiler options and flags. These parameters (or sets thereof) are quite specific to a particular *target platform*; executing the application on a different target platform may require a non-trivial amount of re-building effort (even if the actual application source code is untouched). Hence, applications continue to be executed on the default "home" platform, even if other viable options are present.

Grids and especially clouds present real opportunities for applications to move away from their home environments. If an application run can be obtained in minutes on the Cloud instead of waiting for overnight turnaround times on a local cluster, clouds may be an attractive proposition—provided the monetary and manpower costs are acceptable [38]. In the ADAPT project, we are investigating the feasibility and ease of deploying classes of applications on target platforms other than those on which they normally execute. As a *learning exercise*, we have experimented with a Finite Element Method (FEM) CFD code based on the C++ library LifeV [108], whose home environment is an 128-core cluster, and ported it on other computational platforms: clusters and Amazon EC2 cloud.

3.2 CFD Simulations on Different Platforms

The role of cloud computing as an extension of current HPC capabilities has been evaluated by many researchers. In various scientific fields, the rate of increase of available computing power is closely matched or outpaced by the increase in model complexity and therefore of the requirements for fast, large scale computations—prompting serious consideration of "unlimited, on-demand resources"

that clouds promise; however, this is still controversial [38, 39, 40, 41]. Cloud vendors have been reshaping their services, experimenting with new technologies, and exploring new price policies while users are assessing viability. Several cloud-effectiveness benchmarks have appeared in the literature ([39, 42]). We believe, however, that an assessment of cloud computing as a viable choice in real-life applications requires evaluation of its support for more complex scientific software, as we detail in the next section. The present work also includes early benchmarks of Amazon `cc2.8xlarge` instances, a computational offering that is a candidate to match the performance of traditional computing clusters. Furthermore, most studies focus on time-to-completion; our study takes a broader perspective, including a preliminary assessment of cost aspects [43], and the set of activities required to prepare the execution environment for scientific codes on diverse platforms.

The use of the Cloud as the computational platform for computational fluid dynamics analysis has been explored by several software projects. Among the open source projects we cite CAELinux [109], a Linux distribution including a large set of open source packages for computer-aided engineering (Code_Aster (EDF) [110], Code_Saturne (EDF) [111], Salome (Open CASCADE) [112], OpenFOAM (SGI Corp) [113], and Elmer (CSC) [114]). CAELinux currently supports cloud execution on Amazon EC2 by providing a set of pre-defined virtual machines to be run on the EC2 service. OpenFOAM, an open source package for CFD analysis, can be also executed as a standalone package on the Amazon EC2 [115] computing service and on the SGI Cyclone Technical Computing Cloud. We note here that our work is concerned with *comparing* effort, cost, and issues in executing applications on *multiple target platforms exhibiting secondary heterogeneity* rather than the aspect of porting applications to the cloud.

3.3 Test Applications

Partial differential equations (PDE) are a formidable tool for modeling problems in different fields, ranging from aerospace and automotive, mechanical and structural engineering to biology and biomedicine, ecology, and finance [44]. Explicit and analytical solutions to PDE's of real interest are seldom available and numerical approximations are the norm [45]. FEM is a well established approach to the numerical solution of PDE's [46, 47]. The FEM solution is a piecewise polynomial approximation of the exact one and the differential problem is replaced by an algebraic (linear)

system of equations. The accuracy of the approximation is in general related to the size of each portion ("element") of the computational domain, where the solution is assumed to be polynomial. The finer the reticulation (*mesh*) defining the elements, the larger the algebraic problem to be solved after discretization—and consequently, the computational cost—but the more precise the solution.

3.3.1 First Test Case: Reaction–Diffusion Equation

As a first simple test case, we consider the following PDE in a cubic region for $t > 0$

$$\frac{\partial u}{\partial t} - \frac{1}{t^2}\sum_{i=1}^{3}\frac{\partial^2 u}{\partial x_i^2} - \frac{2}{t}u = -6. \tag{3.1}$$

Boundary and initial conditions are selected in such a way that the exact solution is $u = t^2(x_1^2 + x_2^2 + x_3^2)$ (Figure 3.1). This is generally called a RDE. More details about this test case can be found in [45], Chap. 5. Exact solution is used for checking the mathematical correctness of the code execution.

Since the unknown u in equation (3.1) depends on time t and on the space coordinates x_i, the numerical solution requires both time and space discretization. We use a second order Backward Difference Formula (BDF) for the time derivative and the FEM of order 2 for the space variables. In particular, we use the research C++ library LifeV, developed as a joint project among the Departments of Mathematics at the EPFL, Lausanne, Switzerland, the Politecnico di Milano, Italy, the INRIA in Paris, and our department. The library has been mostly developed for applications of the FEM in blood flow and industrial problems.

3.3.2 Second Test Case: Incompressible Navier–Stokes Equations

Incompressible fluid dynamics represents one of the most challenging, attractive and impactive problems in modern scientific computing. Fast and reliable numerical solutions of NSE—the basic mathematical model for incompressible fluid dynamics—are required in several engineering fields, ranging from automotive/aerospace to geophysical and biomedical engineering [48, 49]. If $[u_1, u_2, u_3]$ denotes the velocity vector and p the pressure of a liquid in the 3-D space with coordinates x_1, x_2, x_3, the incompressible NSE reads

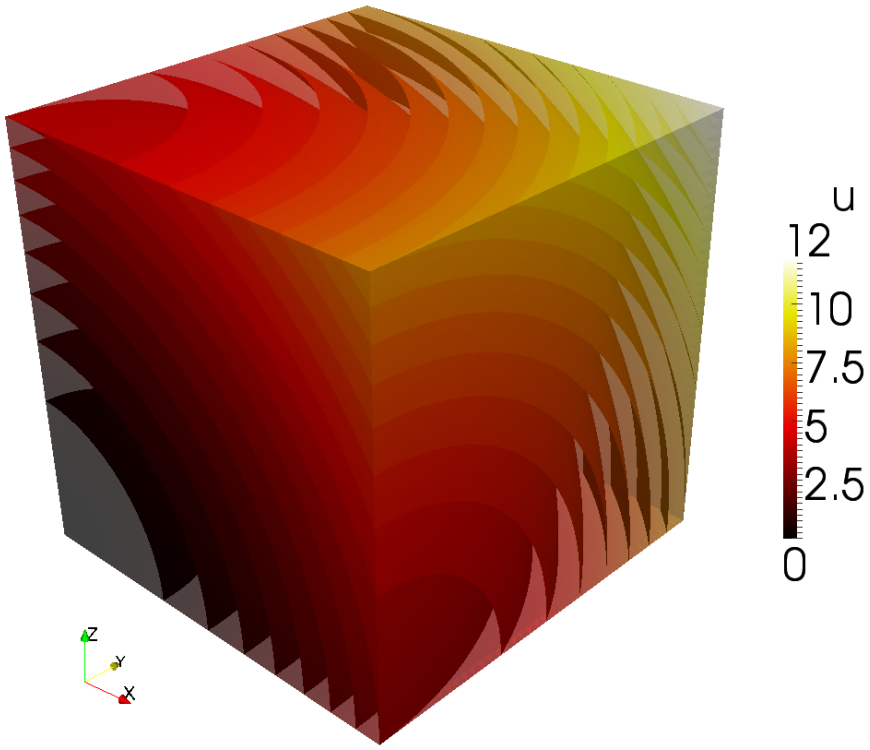


Figure 3.1: Solution of equation (3.1) when $t = 2$s. The isosurfaces of u are plotted inside the cubic domain for a set of 25 values chosen with a constant interval.

$$\begin{aligned} \rho\frac{\partial u_i}{\partial t} - \sum_{j=1}^{3}\left(\frac{\partial}{\partial x_j}\mu\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right) - \rho u_j\frac{\partial u_i}{\partial x_j}\right) \\ +\frac{\partial p}{\partial x_i} = f_i \quad i=1,2,3, \\ \sum_{j=1}^{3}\frac{\partial u_j}{\partial x_j} = 0. \end{aligned} \tag{3.2}$$

Here ρ is the fluid density and μ is the viscosity (that we assume to be constant for simplicity). Vector $[f_1, f_2, f_3]$ is an external forcing term. From the numerical viewpoint, this problem is by far more challenging than RD Equation (3.1), not only due to size (this is a vector problem involving four scalar fields) but to intrinsic mathematical features and the non-linear term (cf. [49]). In particular, we use for our experiments a classical problem proposed by C. R. Ethier and D. A. Steinman [50]—a popular non-trivial benchmark for CFD solvers. The time derivative is discretized with a second order BDF, while the unknowns u and p are approximated using the FEM of order 2 and of order 1, respectively. A snapshot of the exact solution of this problem is shown in Figure 3.2.

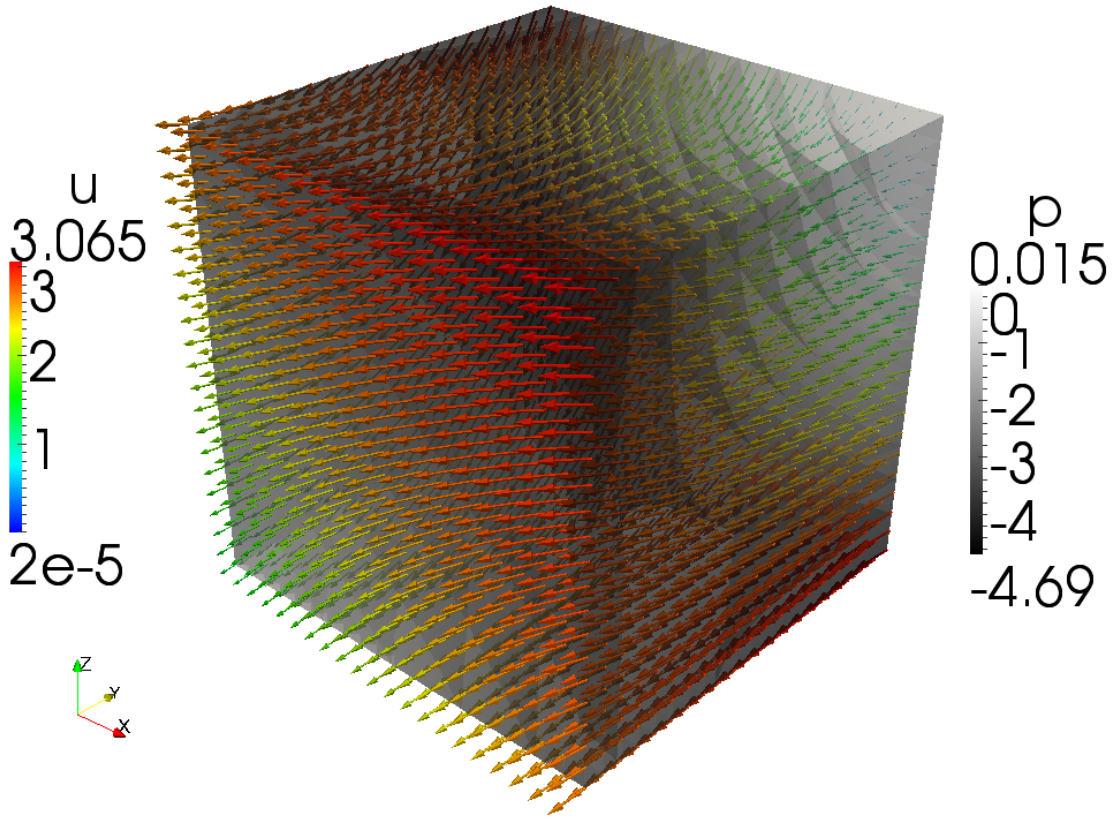


Figure 3.2: Solution of the problem proposed by C. R. Ethier and D. A. Steinman [50], based on Equation (3.2), when $t = 0.003$s. Arrows represent the vector field u, while in the cubic domain are shown isosurfaces of the scalar field p.

3.3.3 The Organization of the Program

The numerical solution of problems like the proposed test cases involves operations that are conceptually split into two categories. The evolution in time is solved as a sequence of steps that compute the unknowns at selected instants t^k. Some operations are independent of the time advancing and are performed out of the temporal loop. Other operations need to be performed at each time step. These typically constitute the computationally intensive kernel of the software. Schematically, we represent the stages of the application as in Figure 3.3.

Here we detail each phase. Step (i) consists of the definition of the *computational domain* where the equations have to be solved numerically. This is given by the mesh. This task is typically accomplished with in-house mesh generators (for structured meshes) or third-party software such as NetGen [116] or GMSH [51]. In a parallel application, the domain is *partitioned* so that each

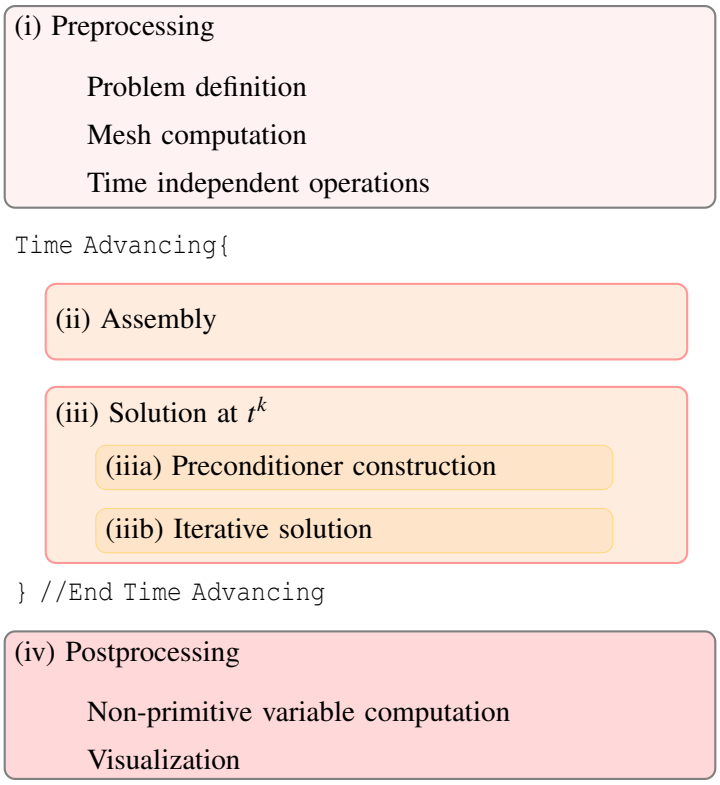


Figure 3.3: Steps for the numerical solution of a time-dependent PDE problem

process takes care only of a subset of the global mesh. This splitting is achieved by resorting to graph partitioning algorithms, such as those implemented in the library ParMETIS [117], guaranteeing a proper load balancing among processes. The load is measured as the number of mesh elements assigned to each process. Other operations of this step refer to all the computations that are time independent and can be performed once.

Step (ii) concerns the computation (or more specifically the *assembly*) of the matrices and vectors required for the construction of the discretized algebraic problem. This is carried out with algorithms and data structures provided by LifeV. In a parallel application, each process can only access a subset of the matrices and vectors, corresponding to its own portion of the mesh. In other terms, matrices and vectors are distributed and need to be updated via a message passing interface. Our software uses distributed data structures implemented in the external library Trilinos [118] developed by Sandia Labs. Trilinos also provides algorithms for the solution of the algebraic problem (Step (iii)). In particular, we use *iterative preconditioned methods*, where the solution of the large linear systems assembled at each time step is replaced by the iterative solution of simpler

systems (called *preconditioners*). For this reason, we distinguish the Step (iiia) for the computation of the preconditioner, and Step (iiib) for the actual solution of the preconditioned system.

Step (iv) concerns the visualization of the solution to the differential problem and can be delegated to third party software such as Paraview [119]. This step may also include the computation of quantities of interest related to the solution u.

We are mostly concerned with Steps (ii) and (iii) as they have a major impact on the entire computational cost of the application.

3.4 Deployment Experiences

In this section we report experiences from the deployment phase for the test applications. First, we give information about the software requirements of simulation programs and, then, we characterize used machines with focus on their heterogeneity aspects.

3.4.1 Summary of the Packages Used in LifeV

The complete list of required packages to build our PDE solver follows:

- LifeV library [108], for the formulation of the algebraic counterparts to differential problems; this library is used directly by the solver application;
- Third-party scientific libraries:
  - Trilinos [118] for the solution of linear systems (data structures and algorithms);
  - ParMETIS [117], used for mesh partitioning;
  - SuiteSparse [120], as a support library extending the capabilities of Trilinos;
  - ANN library [121] for nearest neighbor searching in meshes;
  - BLAS/LAPACK libraries (generic or vendor-specific implementations);
- General purpose and communication libraries:
  - Boost C++ libraries [122], mainly used for effective memory management (smart pointers);

  - HDF5 [123], for the storage of large data on file. For compatibility issues, this package has to be built with the 1.6 version interface;
  - MPI libraries (e.g., Open MPI);

- Compilers:
  - C++ compiler (e.g., GCC version 4 or above);
  - [optional] Fortran compiler, compatible with C++;

- Deployment tools:
  - GNU make;
  - Autotools;
  - CMake (version 2.8 or above).

3.4.2 Four Heterogeneous Target Platforms

In this particular study, we benchmarked two applications for the numerical solution of the two test cases presented in the previous section on four different computing architectures.

As the starting point for our analyses, we selected an in-house computing cluster constituting a computational test bed for the LifeV developer team.[1] As the second architecture, we used a larger compute cluster provided on a fee-for-use basis within our university. Next, we evaluated the usability of on-demand resources provided by Amazon's Elastic Compute Cloud (EC2). From the rich resource offerings provided by this vendor, we picked the most powerful hosts, dubbed *Cluster Compute*. The fourth platform was the HPC supercomputer available for scientists at the CILEA supercomputing center, in Segrate (Milano), Italy—this exemplifies canonical grid usage.

The four platforms are heterogeneous in many respects: they differ in availability (measured as wait time to obtain access to the machine), access modality (privileged vs. unprivileged user), storage (e.g., size of user disk space), build (e.g., presence of the compilers and basic build tools), aggregation (e.g., presence of MPI toolsets), and execution (e.g., presence and type of parallel job schedulers). In this section we compare the considered architectures, pointing out differences and similarities. Table 4.1 summarizes the compared features; below we note a few relevant details.

[1] This is the "home" environment where the application is run by default.

Puma

The in-house computing cluster `puma` comprises thirty two four-core nodes. Each node includes two AMD 2214 processors, 8GB memory with 80GB local scratch disk space, while Gigabit Ethernet (1GbE) provides the network interconnections. This cluster is controlled by Linux CentOS 5.2, Rocks 5.1, and Portable Batch System (PBS) Torque 2.3.6. Users have unprivileged access to the machine, so they need to install any needed software (libraries etc) in user space. As the "home" environment for LifeV developers, this cluster was pre-provisioned with the entire set of packages required to run LifeV-based CFD simulations. Being an internal resource, with restricted user access, `puma` does not implement a monetary accounting system for computing resource usage.

Ellipse

The university cluster `ellipse` consists of 256 four-core nodes with AMD 2218 processors and 8GB RAM; Gigabit Ethernet provides the interconnection fabric. All nodes are controlled by CentOS 4.5. Job execution is performed by the Sun Grid Engine (SGE) 6.1 scheduler which was configured to manage serial processing batches only. As with `puma` users, `ellipse` users have unprivileged access to the machine. The required software dependencies were not originally installed on the cluster. They were provisioned by building them from sources in user space. All users pay a flat rate 5¢ per CPU core per hour.

Lagrange

Our third test architecture was the supercomputer cluster `lagrange` at the CILEA supercomputer center. This supercomputer, when assembled, was placed at the 136^{th} position in the TOP500 list [124]. The machine is composed of HP ProLiant server blades with two Intel Xeon X5660 processors and 24 GB RAM each. The network infrastructure is provided by InfiniBand (IB) 4X Double Data Rate (DDR, 20 Gb/s bandwidth). Computing nodes are controlled by the CentOS version 5.6 operating system. Users have unprivileged access to the machine. However, unlike `puma` and `ellipse`, `lagrange` provides some dependencies for LifeV-based applications (in particular the vendor-specific BLAS/LAPACK package). The cluster runs PBS Professional version 11 as a scheduler. The cost of the computer is €0.15 per core per hour (currently, about $0.20).

EC2

Our final target architecture was a infrastructure as a service (IaaS) cloud offered by Amazon EC2. IaaS resources provide on-demand computing in the form of computing chunks virtualized from the vendor's multitenant machines. These chunks are delivered for users as standard ssh-able root-accessible computational hosts. Users requesting these chunks specify the quantity of hosts, a resource class (characterized by computational power, number of CPU cores, memory capacity, and network interconnect) and the Operating System (OS) controlling the hosts (from *public* or users' *private* OS images). The vendor offers several sizes of virtualized hosts, ranging from small instances (`t1.micro`/`m1.small`; one 32bit CPU, below 1GB of RAM, and slow network interconnections) to modern HPC-class cluster nodes (`cc2.8xlarge`/`cg1.4xlarge`; 16 cores, 60GB of RAM, 2 GPGPU processors, and 10 Gigabit Ethernet (10GbE), with network-aware host allocation strategy—*placement groups*). All setup conditions, as well as management and monitoring measures can be controlled by users in various ways, including direct interactions with the AWS (Amazon Web Services) Management Console web toolkit or Amazon EC2 API command-line tools [125], programming libraries [126], or frameworks providing higher level services over IaaS clouds [34, 127]. In contrast to conventional computational resources, EC2 users obtain full access to hosts instantiated on the Amazon's service. As a result, we could use a system package management tool (`yum`, in our case) and modify the system configuration.

Amazon does not levy any upfront costs and charges users merely for the actual use of resources (time and computational power), external data transfers, and scratch space (size); however, some OS images and additional services (e.g., static IPs) incur additional costs. In this study, we focused on evaluating the `cc2.8xlarge` instance.

3.4.3 Porting Experiences

Execution of our two test applications on the target architectures requires (1) providing all software dependencies, (2) running the actual build program (make) that links against the appropriate libraries and produces the final executable file, and (3) providing the parallel execution environment.

A goal of this exercise was to keep the porting effort to the absolute minimum possible. Thus, no changes were made to the application source codes. We utilized all compatible software that

	puma	ellipse	lagrange	ec2
cpu arch. # cpu/cores	Opteron 2/2	Opteron 2/2	Xeon 2/6	Xeon 2/8
RAM/core	1GB	1GB	1.3GB	3.8GB
network	1GbE	1GbE	IB 4X DDR	10GbE
storage	OK	insufficient disk quota	OK	insufficient image mod
access	user space	user space	user space	root
support	full	very limited	limited	none
build env.	yes	yes	yes	none yum
compiler	GCC 4.3.4	GCC 4.1.2	GCC 4.1.2	none yum
dependencies	all	none src. install.	blas, lapack src. install.	none src. install.
MPI	Open MPI	none src. install.	Open MPI	none yum
parallel jobs	yes	no	yes	no
execution	PBS	SGE	PBS	shell

Table 3.1: Specification of the test architectures differences. In color: how the missing capabilities were addressed.

was already available on the target (even if it was not the latest version) and resorted to installation (preferably from package repositories) only if the dependency was missing or incompatible. In the ec2 case, we had to commit a minimal configuration allowing aggregation of computational hosts for a single parallel execution.

Table 4.1 shows, in brief, the state of capabilities provided by the test resources before porting. Below, we provide a full report describing all the activities required to elevate the resource capabilities to the LifeV build and execution environment.

Puma

This computer fully sustains the build and execution of LifeV-based applications. As the result, we needed to use a generic Makefile to create the executable. To launch the simulations, we used the PBS job submission command.

Ellipse

The `ellipse` environment already provided the GNU compiler collection in a compatible version (4.1.2) with C, C++, and Fortran compilers, as well as all needed deployment toolkits. We began assembling dependencies by provisioning the MPI package (Open MPI 1.4.4). Then, MPI tools were used to build ParMETIS 3.1.1, HDF5 1.8.7, Trilinos 10.6.4, and SuiteSparse 3.6.1. Additionally, we provisioned the Boost libraries 1.47. For the BLAS/LAPACK package we resorted to CPU vendor-specific implementation, available as ACML [128] 4.0.1. The last step was updating the `Makefile` for the simulation applications and building them. All software preconditioning actions took about 8 man-hours of work by an experienced member of the LifeV developers team.

The SGE on `ellipse` was not configured to support parallel tasks; however, Open MPI could detect and liaise with SGE to start and end tasks on assigned nodes. Thus we were able to use SGE commands to reserve and submit `mpiexec` jobs.

Lagrange

CILEA presented a pre-prepared environment for building and executing parallel, MPI-based applications. The administrators provided a choice of C++ and Fortran compilers (GCC version 4.1.2 and Intel Compiler Suites 12.1); MPI packages (Open MPI, Intel MPI), and BLAS/LAPACK routines were available from the CPU vendor-specific libraries (MKL [129]). In order to provision the software dependencies for our software, we used GCC to build the Boost libraries 1.47 and SuiteSparse 3.6.1. The remaining software dependencies (HDF5 1.8.7, ParMETIS 3.1.1, Trilinos 10.6.4, LifeV 2.0.0) were built against Intel MPI compiler wrappers. All the preparatory actions took about 8 man-hours for the LifeV developer.

EC2

To exercise the port of our software to EC2, we initially selected the `cc1.4xlarge` instance (when we started our experiments `cc2.8xlarge` was not available) and the *EC2 CentOS 5.4 HVM AMI* (`ami-7ea24a17`) image. To facilitate software preconditioning steps we used the `root` access. As this version of CentOS Linux contained obsolete versions of software, we began with an update of the system using the `yum update` command. The chosen image contains only the essential

packages, with neither development software nor scientific library support. In order to provide the source code build environments, we installed, using `yum`, GCC 4.4.5, GFortran 4.4.4, libtool 1.5.22 (with autoconf 2.59, automake 1.9.6), and Open MPI 1.4.4. To install CMake 2.8 we resorted to a source code installation as the required version was not available from the repositories. After this phase, we downloaded all required dependencies as the source codes, built, and installed them: GotoBLAS2 1.13, LAPACK 3.3.1, Boost 1.47, HDF5 1.8.7, ParMETIS 3.2.0, SuiteSparse 3.6.1, Trilinos 10.6.4, and LifeV 2.0.0. After these preconditioning steps, building the simulation application was straightforward.

We also encountered cloud-specific issues not seen on traditional resources. One concerned ssh host mutual authentication to enable automatic launch of remote MPI processes by `mpiexec` requiring pre-generation and storage of keys. The second issue was related to configuration of the EC2 service. We modified the *security group* by enabling all intranet TCP ports to allow MPI processes intercommunication. Additionally, we required more disc space for staging the problem meshes (originally, the utilized image provided 20GB partitions). We could fulfill this requirement by instantiating the NFS service or using the *Elastic Block Store volumes* with copies of the files (one volume may be mounted to a single EC2 instance only). However, we decided to increase the size of the original boot partition, consequently supplying the input files from the same volume.

All the changes committed on the running instance can be preserved by creating a private image stored on the Amazon service. This image, in turn, may be used to launch several identical copies of the instance. Such on-demand hosts behave like cluster nodes. Further conditioning may provide a high-availability computing cluster with services such as monitoring or automatic checkpointing. However, we prepared an image that contains merely the essential software packages and services that allow the on-demand resource to sustain our CFD simulations.

In order to execute a simulation, we instantiated an appropriate number of copies of the prepared image. The service assigned intranet IP addresses for the on-demand hosts and we used these IPs to create the run-specific hosts list for the `mpiexec` command. Finally, this command was executed directly from the command line.

3.5 Performance Evaluation

As mentioned, we benchmarked the four described architectures using two test cases: a simple RD test with boundary conditions specifying the exact solution on the boundaries; and a solution of the Navier–Stokes problem where, again, we prescribe the exact solution on the boundary.

3.5.1 RD Test

We executed the simple RD 3-D problem on four computing architectures: two in-home clusters (`puma`, `ellipse`), the HPC-class computer (`lagrange`), and the on-demand instantiated Amazon's hosts (`ec2`). In the case of EC2 resources, we utilized the newly introduced, most powerful Amazon's instance driven by two eight-core Intel Xeon E5 processors with 60.5GB of RAM (`cc2.8xlarge`). Although this instance type was different from the build target, the transition was streamlined—the preconditioned image was fully compatible with both types and the compilers used generated optimized, binary compatible executables. As this node type includes sixteen computing cores, it allowed us to conduct our experiment with a 200^3 element input mesh on 10^3 MPI processes on just 63 instances.

During the execution phase on `ellipse` and `lagrange`, we encountered system difficulties that limited our experiments on these targets. The former machine was not natively configured to execute the parallel jobs and our tasks spanning above 512 processes could not be launched (`mpiexec` was unable to initialize a huge number of remote MPI daemons). On the latter target, our simulation codes reached the configured limit of data volume sent by the IB network adapters. As a result, we could not execute tasks bigger than 343 processes there.

In Figure 3.4, we present results from a weak scaling test of the RD application. We started from a single process and incremented the number of processes as cubic powers to the limits of the platform in question. Simultaneously, we adjusted the problem size by providing more detailed 3-D mesh so every process during the experiment was loaded with 20^3 mesh elements. We recorded iteration wall-clock times across the entire MPI execution: the average times of assembly, preconditioning, and solver phases with the total iteration time. We discarded timings from the first 5 iterations to guarantee that the acquired results are not influenced by Open MPI startup artifacts. Finally, all the consecutive measurements were averaged and are presented in the chart.

As shown in the figure, the problem scales well for all targets in the range 1–125 MPI processes—when the problem size is increased with the machine size, execution times remain reasonably steady (perfect weak scaling would result in constant times). We assume that network performance is the major factor degrading performance in the larger cases—as the problem size grows, processes exchange more data and the overall performance drops. After a certain problem size, only the HPC machine `lagrange` maintains a good weak scaling characteristic. However, we need to investigate why the solver phase on `lagrange` performs better for an increasing number of processes; we think that the placement of nodes in the cluster may play an important role in this phenomenon. Another interesting observation is that, in case of Amazon's hosts, there are certain sizes where the performance significantly deteriorates. These break points depend on the volume of data exchanged in each phase of the simulation. One more striking aspect is that the `ec2` configuration is characterized by the worse performance degradation in comparison to `puma` and `ellipse` (both with 1GbE network). Due to the fact that each utilized EC2 instance incorporates sixteen CPU cores, the on-demand assembly exploits notably fewer hosts hence the smaller volume of data is exchanged by the 10GbE network.

3.5.2 Placement Group Benchmark for RD

We also analyzed how the *placement group* setting influenced the performance of on-demand machines. In order to test this, we executed the RD code in two configurations, both utilizing the same `cc2.8xlarge` instances and preconditioned image. The first configuration exploited the fully paid 63-node assembly located in a single placement group, while the second configuration used 63 nodes acquired both from *spot requests* (instances sold for bid prices) and fully paid requests from four different placement groups in the same *availability zone* `us-east-1a`.

Table 3.2 presents the test results: the average total time for a single iteration and its cost in both configurations (during the test, the regular instance cost $2.40 and the spot-requested—54¢, both prices per host per hour). The results show that regular allocation in a single placement group does not introduce any performance benefits despite costing four times as much. Of course, the unpredictable nature of spot requests makes it impossible to estimate when instances start, how long they are available, and their actual price (although a maximum can be specified). Indeed, we never succeeded in establishing a full 63-host configuration of spot request instances.

# of mpi	#	full		mix	
		time[s]	real cost[$]	time[s]	est. cost[$]
1	1	4.83	0.0032	4.77	0.0007
8	1	5.83	0.0039	5.78	0.0009
27	2	7.28	0.0097	7.58	0.0023
64	4	8.69	0.0232	8.82	0.0053
125	8	21.65	0.1155	21.24	0.0255
216	14	31.47	0.2937	31.47	0.0661
343	22	66.34	0.9729	62.57	0.2065
512	32	92.20	1.9670	94.52	0.4537
729	46	127.76	3.9179	128.10	0.8839
1,000	63	162.09	6.8077	148.98	1.4079

Table 3.2: Comparison of two EC2 `cc2.8xlarge` assemblies: fully paid instances in a single placement group (*full*) and spot requests in various placement groups (*mix*)

3.5.3 Navier–Stokes Test

In Figure 3.5, we present the weak scaling results achieved on our four basic test architectures, using the second application—Navier–Stokes 3-D simulation. We loaded computers as in the first case—every MPI process held 20^3 elements of the input mesh. As with RD, we could not execute this test on all available cores on `ellipse` and `lagrange`. As before, we also discarded the first few iterations to insulate the timings from MPI startup impact; the chart presents averaged times for all observed iterations.

The Navier–Stokes test is more computationally demanding than the simple RD test. Moreover, the data volume exchanged among the MPI processes during the computation increases as this problem involves two variables. This test does not scale well in any range; however, again the most efficient machine is the HPC `lagrange` cluster. We believe that the results manifest the obvious explanation, i.e. that this type of CFD simulation is critically dependent on network performance. Again, the performance of Amazon cluster nodes declines sharply as the problem size/number of processes increases. However, for computationally intensive tasks for a small number of processes, Amazon EC2 performance is comparable to the HPC class machine and can considerably improve time to completion in comparison to the department class computing clusters.

3.5.4 Cost Analysis

Figures 3.6 and 3.7 compare costs for resource utilization. We estimate the cost of our department cluster `puma`, based on its real capital cost and operating expenses, at 2.3¢ per core-hour, which is consistent with other published estimations [52]. For our university cluster `ellipse`, users pay a flat rate of 5¢ per core-hour. The cost per core for the 16-core EC2 instances applied in the study starts from 3.375¢, if spot requests are used (`ec2` spots), or 15¢ for flat-price nodes (`ec2` regular). However, as Amazon charges the users for the entire machine, this price increases if not all cores are utilized, as shown on both charts for two first cases. Finally, the cost of `lagrange` was set at 19.19¢ per core-hour based on the prevailing currency exchange rate.

Perhaps unsurprisingly, compute-intensive applications are most cost effective—one obvious reason being that neither clouds nor grids charge for network utilization and all charges are based on nodes. This is readily apparent in the case of the Navier–Stokes application—EC2 costs less than our on-premise cluster and is faster as well. Both figures contain the "`ec2` mix" curves which could be viewed as a *cost-aware strategy* for Amazon's resources. However, obtaining a large number of hosts via spot requests is difficult if not impossible at all. In our experiments, we were compelled to add regularly-priced hosts to spot-request hosts to obtain the size configuration needed; this is apparent in the convergence of the mix and regular curves.

3.6 Experiences and Additional Research

We have presented preliminary experiences and observations based on our exercise to deploy two production CFD codes on four different target platforms characterized by heterogeneity in secondary attributes. Noteworthy is the *effort required* for preparing the target platforms for the execution including provisioning of the application, required packages and libraries installation and other logistical hurdles. In this study, we softconditioned all machines manually and installed only the necessary and sufficient packages. We observed that provisioning of a machine took about a day for an experienced member of the development team; in addition, multiple requests and interactions with system administrators were needed. It significantly hampers switching between machines. `ADAPT` project aims at unsupervised software deployment and in Chapter 6 we show how this mundane and unproductive task can be handle nimbly and automatically.

Comparing on-premise and on-demand targets for the applications we tested, *we found some evidence to support the claim that IaaS resources may be utilized for scientific CFD simulations possibly at lower cost than incurred locally*. In particular, the spot-request feature coupled with availability of cutting edge resources (16-core nodes, 60GB RAM as opposed to 3-year old, 2–4 core nodes with 4GB RAM), suggests that small on-demand assemblies may be a viable alternative to local clusters. It is not without significance that IaaS's provide resources immediately, while local and grid resources are often subject to long queue wait times—an aspect that might offset any additional expense. Another factor is size; at least in our case, only Cloud providers could provide a large enough offering to sustain the biggest, 1000-core task. Furthermore, while a modern local computing cluster, with an efficient interconnection network will outperform an on-demand assembly (which is highly vulnerable to network performance), the cloud solution might be useful for other reasons.

The cost and performance considerations open other research areas. The first extension should explore possible performance tuning. The SaE applications are by-design developed for HPC platforms and this assumption does not have to work well on other platforms such as clouds; investigation related to performance adaptation is presented in Chapter 4. Another research topic is inspired by utility computing ideas: if an application can be executed on various targets that have different performance and cost characteristics, which platform should be used? The tradeoffs between time-to-completion and cost are deeper studied in Chapter 5.

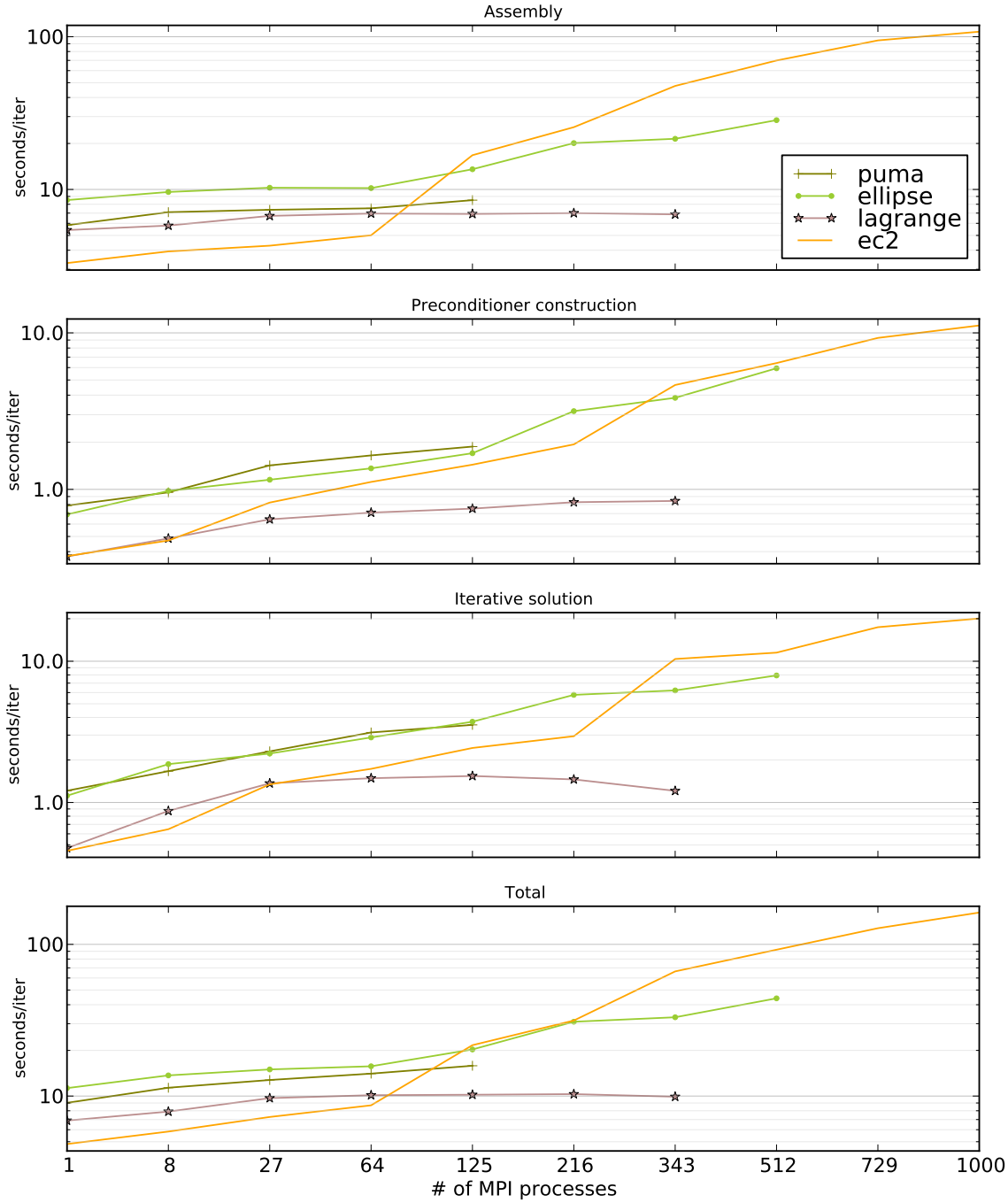


Figure 3.4: The weak scaling test of the RD 3-D simulation. The initial size of the problem mesh is 20^3.

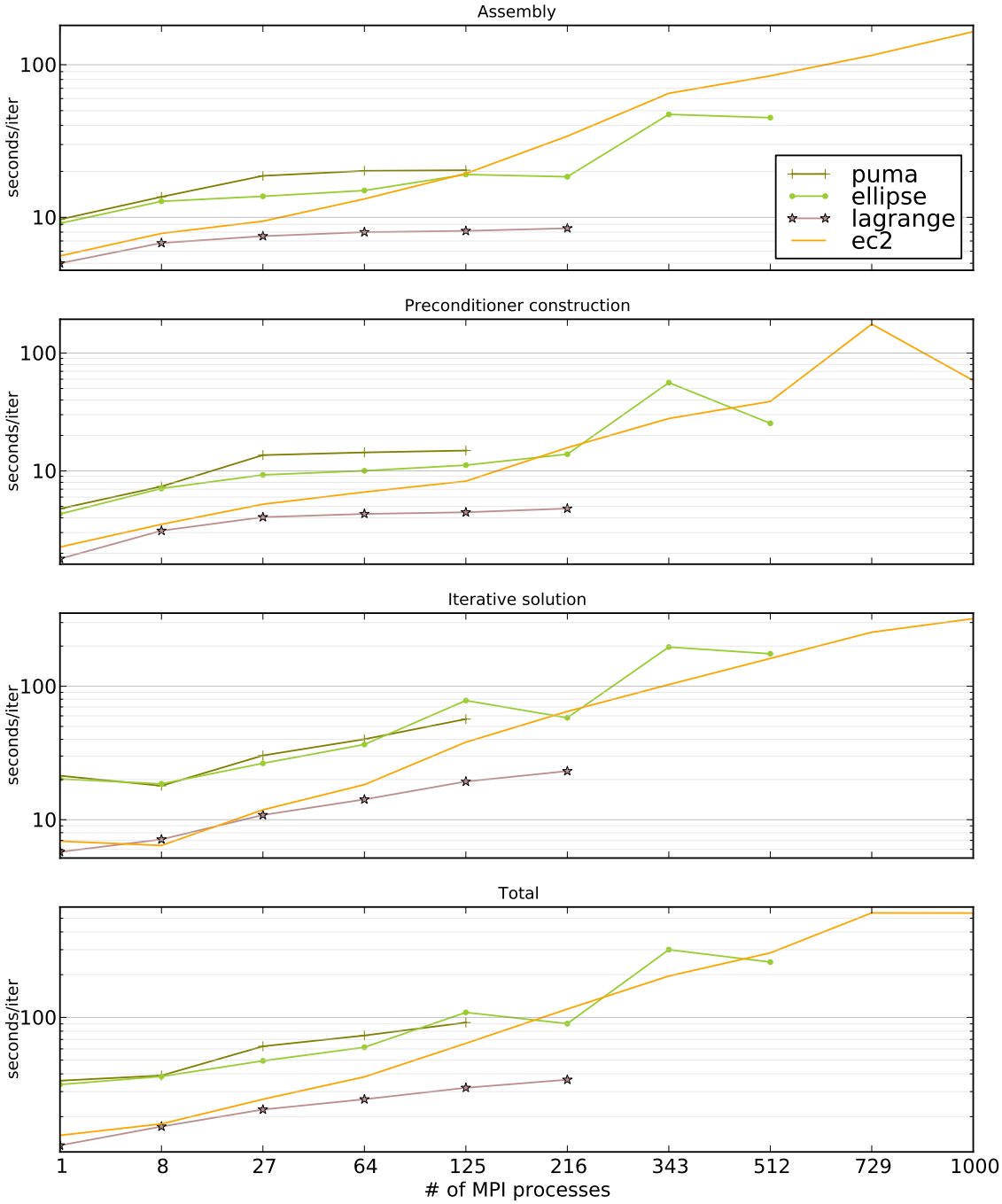


Figure 3.5: The weak scaling test of the Navier–Stokes 3-D simulation. The initial size of the problem mesh is 20^3.

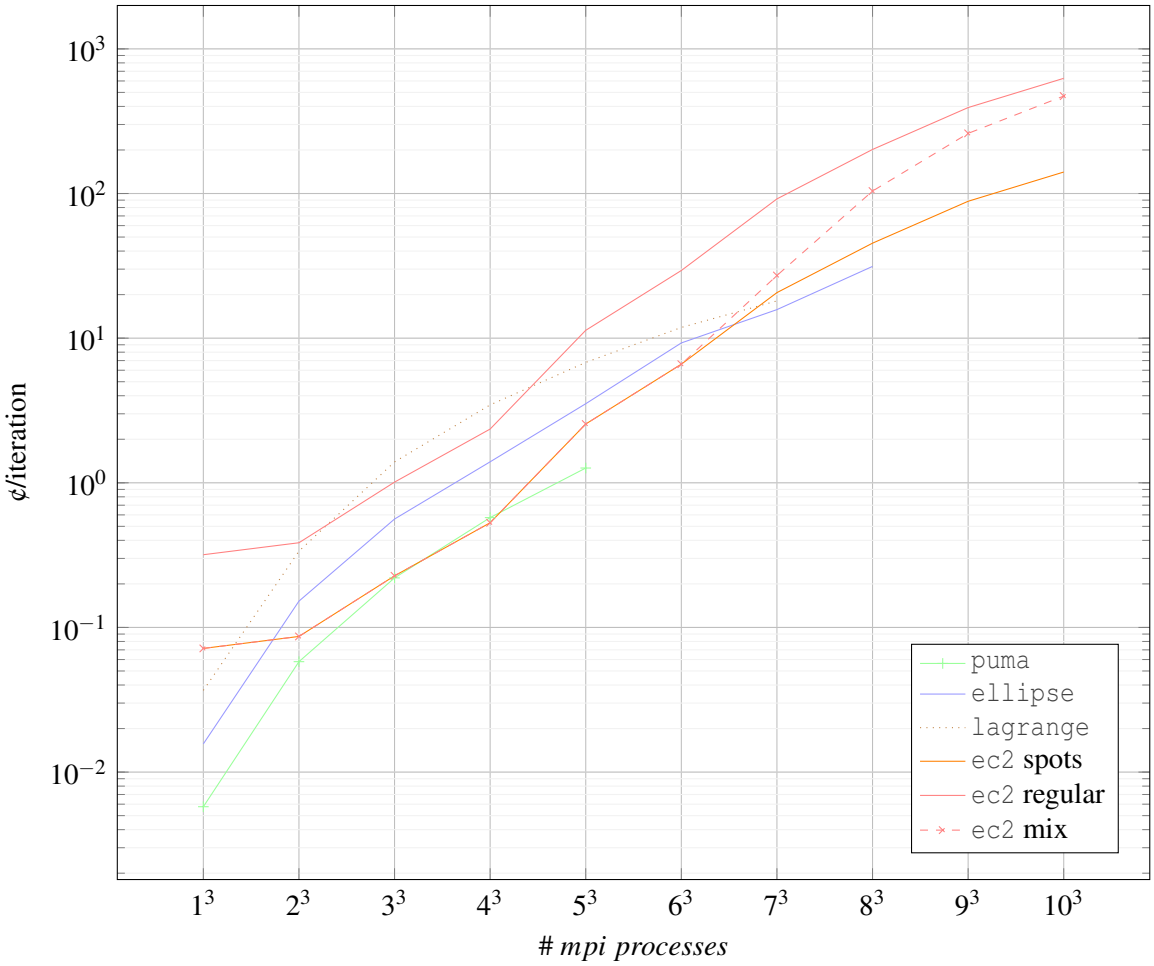


Figure 3.6: Per iteration costs of the test architectures for the RD application weak scaling benchmark

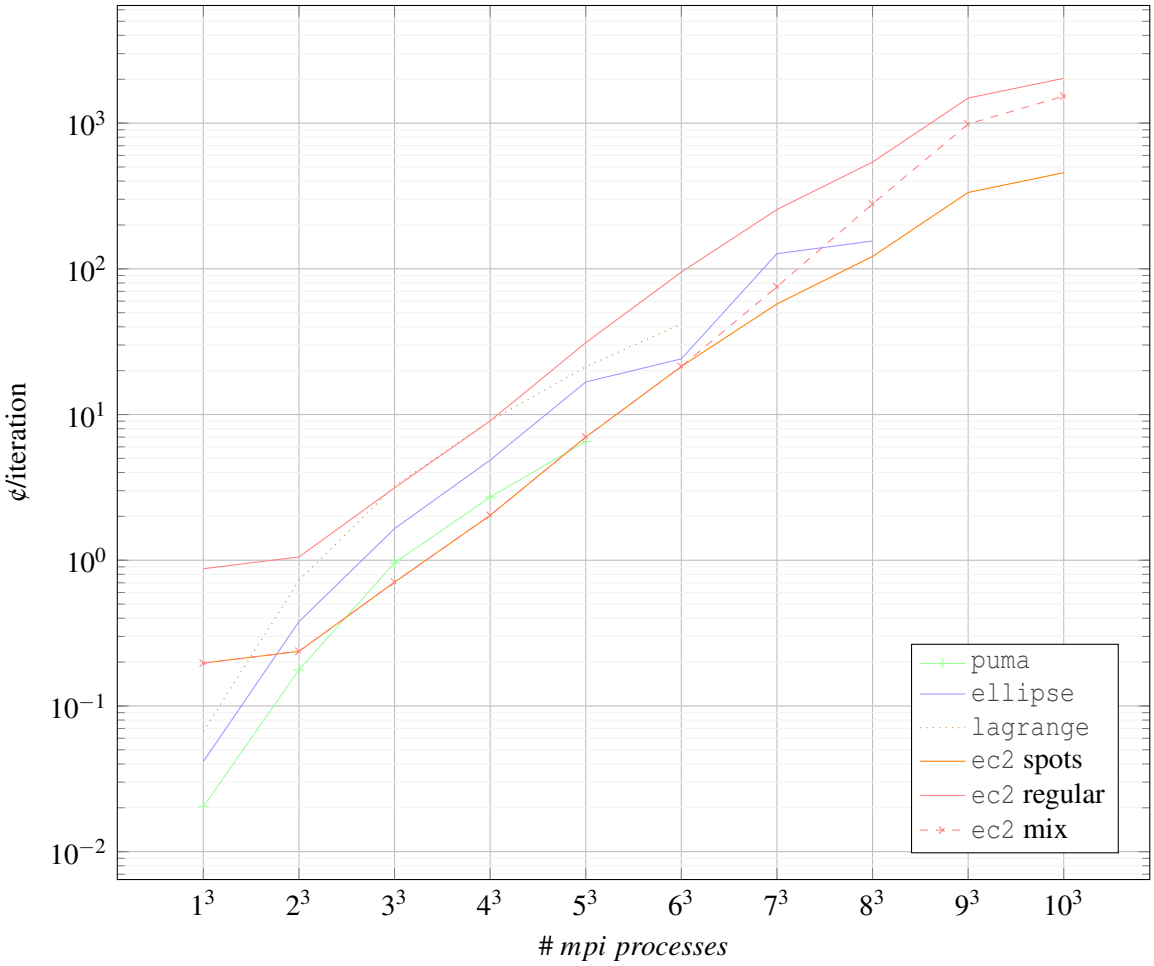


Figure 3.7: Per iteration costs of the test architectures for the Navier–Stokes application weak scaling benchmark

Chapter 4

Parallel MPI Applications Performance Adaptation

4.1 Background

Computing platforms for HPC applications are now expanding from on-premise clusters and grids to IaaS and occasionally PaaS cloud environments. On clouds, a great deal of attention has been devoted to computing efficiency and data handling but there has been relatively little focus on interconnection network capabilities. While several cloud providers have "cluster" offerings, there is little guarantee of QoS as far as communication channels are concerned. Unless carefully controlled, IaaS units may be allocated on different boards, racks, or datacenters. In terms of processing units themselves, CPUs are now universally multicore. For explicit message passing, e.g., MPI, parallel programs that are common in science and engineering, these two factors lead to substantial heterogeneity in communication capabilities—with significant resulting impact on application behavior and performance. Logically, an MPI process communication graph is uniformly fully connected but processes placed on cores within a node can observe communication performance several orders of magnitude better than those physically or geographically distant.

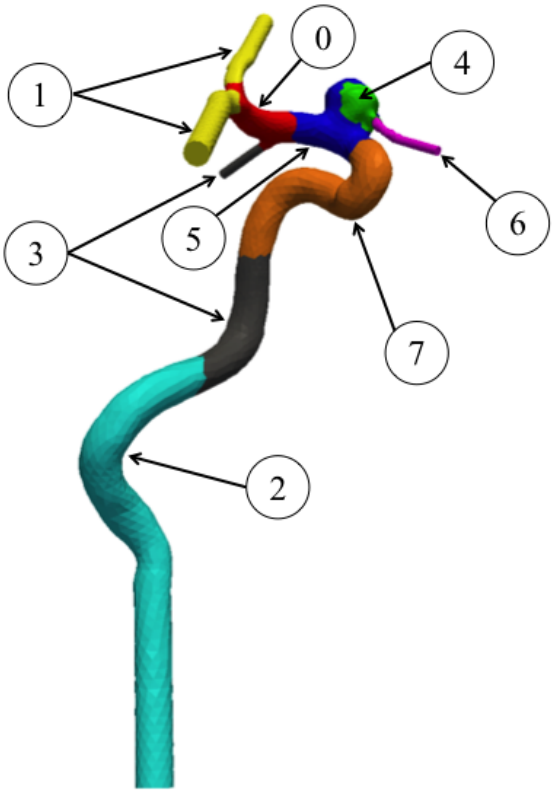
Practically, most real-life applications are not regular or symmetric and thus their MPI process communication graphs are unevenly weighted. In this study, we examine an example SaE hemodynamic simulation based on LifeV that is representative of a large class of numerical

simulation applications. This application utilizes mesh generation and partitioning techniques to divide the problem according to the requirements of the physical system being simulated, eventually decomposing the solution to assign approximately equal *computational* work to each process. A side-effect of this partitioning is the heterogeneous nature of the resulting communication pattern. Depending on the nature of the numerical simulation, the domain decomposition techniques used, and problem size parameters, different pairs of MPI processes interact in different patterns. An example is shown in Figure 4.1; in simulating the blood vessel shown in Figure 4.1a on eight processes, the resulting MPI process graphs shown in Figure 4.1b and Figure 4.1c depict major pairwise communication volume and number of messages, respectively, with line thickness representing quantity. It is immediately apparent that given a parallel platform with heterogeneous communication channel capacities, processes that interact more *should logically* be placed along higher capacity, faster links.

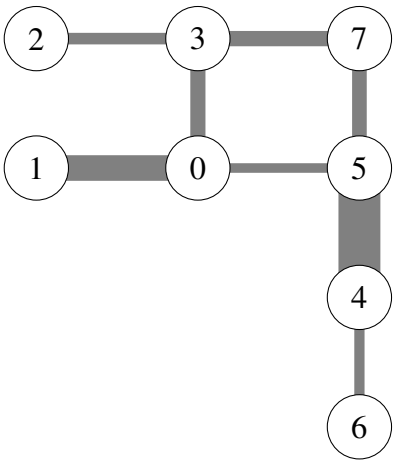
In this study, we investigate the issue of appropriate process placement in MPI programs whose process interaction graphs are heterogeneous, on platforms in which message passing parameters vary between different pairs of processing elements due to the platform/environment architecture. In many cases, it is possible to determine relative communication volumes and frequencies between different pairs of MPI processes and correspondingly, relative communication capabilities between different computing elements in a given target platform. This information may then be used to achieve a best-effort mapping of MPI processes to processor cores that overlays application graph edges on platform graph edges in a weight-aligned manner. We describe detailed experiments with our hemodynamics CFD code in three environments, and discuss several interesting findings that highlight situations in which such an approach can be very effective. In the broader context, this example shows that `ADAPT` can automatically tune the performance of a parallel SaE application based on the predicted communication interactions.

4.2 Mapping Parallel Components into Processing Elements

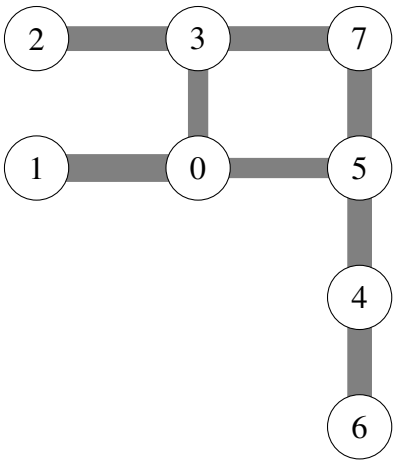
Improving the layout of the tasks of a parallel application on a particular hardware architecture is an attractive research topic as it may increase the performance of the application without requiring modifications to the source code. The advantages may be particularly significant if the supporting



(a) 3-D mesh of the blood vessel



(b) Relative communication volume



(c) Relative message quantity

Figure 4.1: The simulated blood vessel and its exemplary partitioning for parallel processing

computing machines are high-performance clusters with non-uniform interconnection network links, and the optimized placement is able to map to the capabilities of cutting edge network solutions. Several projects attempt to address some or all aspects of this issue. Rubik [130] is a software toolkit that applies simple geometry transformations (e.g., splits, tilts) to the Cartesian task topology of an application, altering its mapping to the Cartesian network topology. Thanks to task-shuffling, the underlying hardware may better support MPI collective operations by utilizing more hardware links while avoiding excessive latency or congestion [53]. Our solution follows a similar approach: we design the layout of MPI tasks *before* we execute the application. However, as our hemodynamic CFD code mainly uses point-to-point communication and has no statically defined communication topology, an analysis of data exchange profiles is needed for each execution and each input of the application in order to determine communication patterns and optimize task mapping.

The process placement mechanism is greatly facilitated by the current versions of MPI frameworks. OpenMPI [131] provides several mechanisms supporting process arrangements, starting from simple, built-in process iterations over the set of available hosts to the process binding maps. Moreover, the modular architecture of OpenMPI allows extending this functionality. Implementation of LAMA (Locality-Aware Mapping Algorithm) that is described in [54] follows the OpenMPI Resource Mapping Subsystem interface extending the execution capabilities of OpenMPI. By applying this component, users can control cache- and NUMA-aware process placement specifying a *process layout* which assigns the sequence of MPI processes with nodes, boards, CPU sockets, cores, and hardware-supported threads. To gain the locality-aware performance benefits, this description should reflect the application communication requirements. However, the communication pattern for our application depends mainly on the discretized representation of the physical geometry (e.g., an artery); consequently, we cannot prescribe its behavior before the input is known. Currently, to set the process placement for a specific run, we use directly names of hosts allocated for the task—using LAMA, we could abstract from maintaining actual resources and focus on the target architecture.

A successful mapping strategy has to consider also the properties of the network backend. Eliminating unnecessary network hops may improve the overall latency and lead to better performance of the executed application. The project described in [55] considers the homogeneous,

multilevel IB network and offers an improved MPI implementation that exploits the network topology to increase intra-node communication and reduce long distance inter-node communication. While this is an eventual goal for our project also, we do not require a different (modified) MPI framework implementation. Moreover, even though we currently consider a simple, single hop network topology, we propose methods that can be extended to different scenarios. In particular, when considering broader network scenarios, a more aggressive mapping can be applied, aggregating machines from geographically separated data centers to provide the computational platform for distributed applications—with the least possible degradation in communication performance. In our companion project we have evaluated the possibility of inter-cloud aggregation and have conducted preliminary measurements of the performance of such integrated systems [17].

4.3 Problem Description

4.3.1 Test Case: Blood Flow in a Cerebral Aneurysm

Computational models based on the NSE have become a valuable tool for the study of blood flow problems due to their cost-effectiveness and flexibility with respect to *in vitro* experimental studies. They allow the analysis of blood flow dynamics in subject-specific vascular geometries and the controlled and reproducible assessment of the effect of experimental parameters such as heart rate, average flow rate, and flow conditions in the neighboring parts of the circulatory system.

The *image-based CFD pipeline* for blood flow problems starts with the image of a part of the circulatory system (typically including one or more arteries) acquired with techniques such as magnetic resonance or 3-D rotational angiography. A geometric model of the vascular structures is extracted from the image by means of segmentation techniques (see e.g. [56]). Such methods aim at characterizing the shape of the vessels, by identifying the contour separating blood from other biological structures in the image. This yields the definition of a surface that represents the domain of interest in which blood flow is simulated. To this end, a three-dimensional mesh is built. The Navier–Stokes equations are solved at some selected instants within the time interval of interest assuming that blood velocity or blood pressure are known on the boundary of the volume of interest. In particular, subject-specific measurements may be used to quantify the unknowns on the *inlet* and *outlet* sections of the vascular geometry, while blood velocity is assumed to be

zero in contact with the vascular wall (*no-slip* condition). This information is used to prescribe boundary conditions, required for the solution of the differential problem. The time discretization is performed by a suitable approximation of time derivatives, using the value of the solution in the collocation instants. Among others possibilities, we use here *Backward Difference Formulas (BDF)* with an error proportional to the square of the distance between two consecutive instants (second order methods) [57, 58].

The model problem used in our experiment is the blood flow in an internal carotid artery affected by a saccular brain aneurysm. Aneurysms are localized dilations of the arterial wall, often in the form of a blood-filled sac. They may rupture causing severe brain damage and even death. Fluid dynamics is considered one of the method that may help predict the outcome of the disease [59]. We consider a subject-specific arterial geometry, extracted from medical images acquired and processed during the multi-center research project Aneurisk [132]. This kind of geometries are available for download through the web portal AneuriskWeb [133], an open-access repository of computational studies on cerebral aneurysms carried out as part of Aneurisk.

To compute blood velocity and pressure in the subject-specific geometry we solve numerically the Navier–Stokes equations, using LifeV (cf. subsection 3.3.3). We simulate blood motion under pulsatile flow conditions, representing the pumping action of the heart. A time-varying flow rate is prescribed in the internal carotid artery reproducing a realistic waveform [60]. Stress-free conditions are prescribed on each outflow section. Blood is described as a Newtonian fluid with density $1\,\mathrm{g\,cm^{-3}}$ and dynamic viscosity $0.035\,\mathrm{dyn/cm^2}$. For the sake of the analysis presented in this manuscript, we limit our simulation to a short time interval ($0.10\,\mathrm{s}$), solving the discretized NS equations at 10 instants (i.e., the simulation time step is $0.01\,\mathrm{s}$). A snapshot of the computed solution is shown in Figure 4.2.

The described problem is proposed in [61] to benchmark the sensitivity of CFD pressure predictions and flow patterns to some particular aspects of the image-based CFD pipeline. The benchmark involves the simulation of blood flow in a giant aneurysm grown in the internal carotid artery. All the physical features are assigned ($\mu = 0.04\,\mathrm{Poise}$ and $\rho = 1\,\mathrm{g\,cm^{-3}}$). We consider a single scenario among the ones proposed in the original benchmark, that is the simulation of a pulsatile flow with a mean flow rate of $5.13\,\mathrm{mL\,s^{-1}}$.

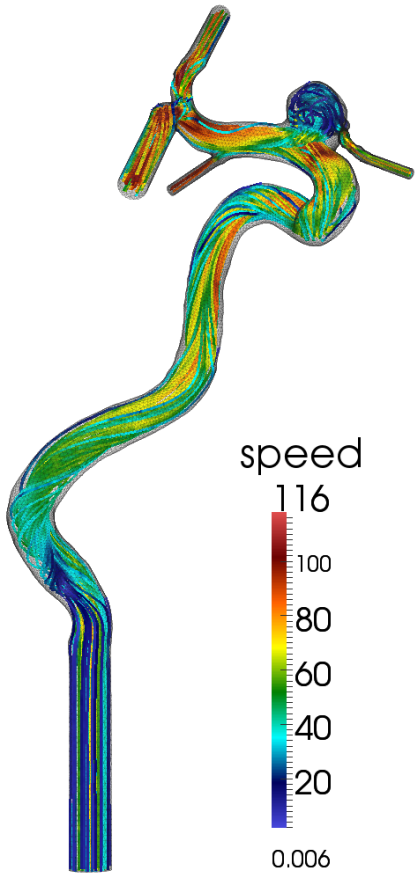


Figure 4.2: Solution of the problem, based on equation (3.2), when $t = 0.05\,\mathrm{s}$. Streamlines of the velocity field colored by the blood speed

4.3.2 Offline Mesh Partitioning

The global mesh consists of the set of all elements, faces, edges and vertices in the tessellation. To each of those entities a unique identifier (*global id*) is assigned. In the following we will denote by N_{el}, N_f, N_{ed}, N_v the total number of elements, faces, edges, and vertices in the global mesh. The topology of the mesh is described by the relationship between different geometric entities and can be expressed in table format (*connectivity tables*):

- the element-to-face table B_0, with size $N_{el} \times N_f$, such that

$$(B_0)_{ij} = \begin{cases} 1 & \text{if face } j \text{ belongs to element } i \\ 0 & \text{otherwise} \end{cases}$$

- the face-to-edge table B_1, with size $N_f \times N_{ed}$, such that

$$(B_1)_{ij} = \begin{cases} 1 & \text{if edge } j \text{ belongs to face } i \\ 0 & \text{otherwise} \end{cases}$$

- the edge-to-vertex table B_2, with size $N_{ed} \times N_v$, such that

$$(B_2)_{ij} = \begin{cases} 1 & \text{if vertex } j \text{ belongs to edge } i \\ 0 & \text{otherwise.} \end{cases}$$

Other connectivity tables can be obtained by composition of the above tables. For example the element to vertex connectivity is given by $B_0 \wedge B_1 \wedge B_2$. Here and in the following we denote by the symbol $\wedge$ the Boolean multiplication operator between tables.

In the parallel application, the computational domain is partitioned in non-overlapping subdomains so that each process takes care of only a subset of the global mesh. In the following we will refer to these subsets as *local meshes*. The splitting is achieved through the use of graph partitioning algorithms, such as those implemented in the libraries ParMETIS [117] or Scotch [134], guaranteeing a proper load balancing among processes. The load is measured as the number of mesh elements assigned to each process.

In our case, local meshes are not overlapping in the sense that each element belongs to one and only one process; however some faces, edges, and vertices are shared among two or more local meshes (*interface entities*). Since the unknowns of the finite element discretization are associated with the entities of the mesh, synchronization between processes is required when accessing the unknowns associated to an interface entity. High quality partitionings should minimize the edge-cut or the number of connections between disjoint partitions. This property is valuable to reduce the communication between processes necessary to synchronize interface unknowns. For large scale simulations, mesh partitioning is a highly memory intensive operation due to the size of the global mesh and it is usually performed *offline* on dedicated machines since many times memory on computational nodes is a limiting resource. In more detail, in our application we considered the following strategy for mesh partitioning:

1. The element adjacency graph $\mathcal{A}$ is built from the topological information stored by the mesh as $\mathcal{A} = B_0 \wedge B_0^T$. The element adjacency graph is an *unweighted* symmetric graph such that two elements of the mesh are connected by a link if they share a common face.

2. The element adjacency graph $\mathcal{A}$ is partitioned in np connected components by using the *recursive bisection* multilevel partitioning algorithm implemented in ParMETIS, where np is the number of desired processes to run the simulation. The result of the partitioning algorithm is a vector p of integer numbers of length N_{el}, in which the value of entry i ($0 \leq i \leq N_{el} - 1$) specifies to which partition element i was assigned.

3. The global mesh is split according to the partitions of the elements induced by p. By introducing the Boolean table $\mathcal{P}$, of size $np \times N_{el}$,

$$\mathcal{P}_{ij} = \begin{cases} 1 & \text{if } p[j] == i \\ 0 & \text{otherwise,} \end{cases}$$

   the local mesh corresponding to process i is associated to its own set of elements, faces, edges, vertices evaluating the non-zeros entries of the i-th row of the matrices $\mathcal{P}_f := \mathcal{P} \wedge B_0$, $\mathcal{P}_{ed} = \mathcal{P}_f \wedge B_1$, $\mathcal{P}_v := \mathcal{P}_{ed} \wedge B_2$, respectively. The above matrices are also used to define proper mappings between the local meshes and the original global mesh, while local connectivities tables B_i^l are obtained by extracting the appropriate rows and columns from the global tables B_i.

4. We finally compute the partition connectivity graph $\mathcal{M}$ that can be used to estimate the communication volume due to synchronization of the variables associated to interface mesh entities (cf. Section 4.4.2). $\mathcal{M}_{ij}$ is proportional to the number of variables shared by processor i and j, i.e., to the number of shared faces, edges and vertices. Thus, we have

$$\mathcal{M} = \alpha_f \mathcal{P}_f \mathcal{P}_f^T + \alpha_{ed} \mathcal{P}_{ed} \mathcal{P}_{ed}^T + \alpha_v \mathcal{P}_v \mathcal{P}_v^T,$$

   where α_f, α_{ed}, α_v are constant values expressing the number of unknowns associated to each face, edge, and vertex, respectively. These constants depend only on the polynomial degree

of the finite element basis. For example, for linear elements we have $\alpha_f = 0$, $\alpha_{ed} = 0$, $\alpha_v = 1$, while for quadratic elements $\alpha_f = 0$, $\alpha_{ed} = 1$, $\alpha_v = 1$.

As a matter of the fact, in the finite element methods the unknowns of the problem are associated to the vertices of the mesh (in the case of linear finite elements). In parallel applications unknowns associated with vertices shared by more than one local meshes represent synchronization points for the applications. Figure 4.3 shows the graph $\mathcal{M}$ for an example mesh divided into eight parts. The vertices of $\mathcal{M}$ represent the parts of the mesh; the weights of $\mathcal{M}$ associated with the edges indicate the number of mesh entities shared by adjacent partitions (vertices, edges, and faces, respectively).

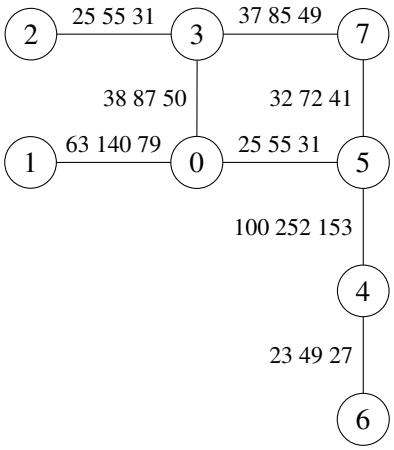


Figure 4.3: Partition of the *coarse* mesh into eight parts

4.3.3 Modeling the Communication Patterns in the Application

In this benchmark test we are mostly concerned with Steps (ii) and (iii) outlined in Figure 3.3 as they have a major impact on the entire computational cost of the application. The finite element matrix assembly phase, Step (ii), is the most embarrassingly parallel kernel in the application since the matrices are first assembled locally and independently by each process. Once local matrices are computed, a synchronization step occurs to collect the local contribution to the global matrix entries relative to interface entities. The amount of data to be transferred during this phase between process i and j is proportional to $\mathcal{M}_{ij}$.

The solution of the algebraic linear system, Step (iii), is more involved and it entails several synchronization steps and the alternation of communication and computation. Iterative methods for

the solution of the linear system consist, at each iteration, of the following three computational kernels: (1) matrix–vector multiplication, (2) vector inner product, and (3) solution of the preconditioner. From the communication point of view, the matrix–vector multiplication entails the same pattern described in Step (ii), the vector inner product requires an all to all communication (`AllReduce`), while the application of the preconditioner has a hardly predictable communication pattern due to load repartition algorithms implemented in Trilinos. To be more specific, as preconditioner we use the algebraic multigrid (AMG) solvers implemented in the Trilinos package ML [62]. AMG builds a hierarchy of agglomerated linear systems, whose size decreases as we move from one level to next. In order to balance between computation and communication, if the number of unknowns associated to process p at level l in the hierarchy falls below a given threshold, then the data structures representing the problem at level l are moved to other processes (by using a load re-balancing algorithm) and processor p is left idle during the computations of all levels below l.

4.4 Evaluation

We executed the hemodynamics simulation using three input use cases. Starting from a single physical model, viz. Figure 4.1a, we generated three meshes that differed in the number of tetrahedral elements. The first, *coarse* mesh includes 31545 elements, the *medium* resolution case consists of 154815 elements, and, finally, the *fine* mesh discretizes the problem into 301580 elements. Then, each mesh was processed by the offline partitioner which generated the local submeshes for the parallel computation for 8, 16, 32, 64, and 128 MPI processes.

In this study, we used three platforms that exhibit heterogeneity in many attributes as the experimental test beds for our benchmarks. Table 4.1 describes and briefly characterizes our platform choices. The goal of our experiments was to investigate how the mapping of MPI processes (*ranks*) to individual processing units (cores) influences the performance of the hemodynamic simulation due to bandwidth and latency of the network. We had two levels of interprocess communication as each tested platform consists of interconnected hosts equipped with multi-way processing units: (1) between computing units in the same node (tens of GB/s bandwidth, nanosecond level latency) and (2) between nodes using network transport (1-10 Gb/s bandwidth, a few microseconds latency). To simplify this study and to avoid introducing other communication

	puma	lab	amazon
description	32-node cluster	desktops in the computing lab	IaaS us-east-1a cc2.8xlarge
cpu cores RAM network	Opteron 2214 2x2 8GB SDR IB	i7-2600 1x4 8GB 1GbE	Xeon E5 2x8 60.5GB 10GbE
OS access exec. env.	Rocks 5.1 user space PBS	Ubuntu user space interactive shell	AMI 12.03 privileged interactive shell

Table 4.1: Specification of the test architectures

distinctions, we restricted our analyses to these criteria (i.e., we disregarded attributes such as affinity zones, hierarchy of cash, inernetwork structure).

To harness the highly imbalanced pairwise communication pattern of our application (cf. Figure 4.1), process placement should exploit communication characteristics to allocate highly coupled MPI processes (to different cores) on a single node to *maximize* their interaction while *minimizing* the utilization of the interconnection network between MPI processes on different nodes. In the ideal scenario, all processes placed on a single node should be mutually exchanging large volume of data, while processes exchanging small volume of data may be placed on distinct nodes.

4.4.1 Message Passing Patterns

We evaluated the *quantities* and *volumes* of the exchanged messages between MPI processes using the tracing ability of the TAU Performance Analysis System [63]. In order to extract message statistics, we ran each instance of our application with TAU. As output, this tool reported *communication matrices* for point-to-point and collective MPI calls. Figure 4.4 shows the distribution of the quantity and volumes of messages passing between processes reported by TAU for the case from Figure 4.3, viz., the *coarse* mesh.

To simplify the task of mapping processes to cores, we focused entirely on the volume of the point-to-point messages. As TAU revealed the message statistics for each pair of MPI processes in both directions separately, we summed these two values to obtain a single measure for the communication link. Such data was used to generate (undirected) communication graphs. In the following, we will denote the volume communication graph as $\mathcal{D}$.

@	0	1	2	3	4	5	6	7
0	0.0	104.5	40.3	76.3	40.3	76.2	12.3	12.3
1	104.8	0.0	1.1	29.6	0.8	29.5	0.6	0.7
2	40.3	1.1	0.0	93.3	0.8	1.2	29.4	1.2
3	76.6	29.5	93.5	0.0	0.7	1.6	1.2	93.5
4	40.3	0.8	0.8	0.6	0.0	92.9	92.8	0.7
5	76.4	29.5	1.2	1.6	93.1	0.0	1.2	93.4
6	12.5	0.7	29.4	1.3	92.9	1.2	0.0	29.0
7	12.3	0.5	1.2	93.7	0.8	93.6	28.9	0.0

(a) Number of exchanged messages [in thousands]

@	0	1	2	3	4	5	6	7
0	0.0	84.5	6.1	53.6	6.0	38.7	5.2	5.6
1	145.2	0.0	3.6	3.8	3.5	4.0	3.1	3.3
2	6.0	3.1	0.0	32.6	3.7	3.7	3.7	3.4
3	82.2	3.8	71.2	0.0	3.3	4.1	3.5	44.5
4	6.3	3.5	3.3	3.3	0.0	120.0	29.8	3.3
5	47.2	4.0	3.6	3.5	256.2	0.0	3.3	41.5
6	4.9	2.7	3.6	2.4	62.1	2.9	0.0	3.2
7	5.2	3.2	3.0	92.9	3.6	89.9	3.8	0.0

(b) Volume of exchanged messages [in MiB]

Figure 4.4: Message passing patterns for the *coarse* mesh; the eight processes case

4.4.2 Correlation of Data Exchange with Partitioning

We compared the communication graph $\mathcal{D}$ with its corresponding partitioning graph $\mathcal{M}$ generated by the partitioner and observed, as expected, a close correlation: the more shared entities between partitions (the larger the weight $\mathcal{M}_{ij}$), the more data is exchanged by the processes i and j. We applied a regression model to discover how accurately the volume of exchanged data can be predicted based on data provided by the partitioner.

More specifically, by letting β_0 and β_1 be the regression coefficients, we build an estimator $\mathcal{H} \simeq \mathcal{D}$ such that $\mathcal{H} = \beta_0 \mathcal{W} + \beta_1 \mathcal{M} + \mathcal{E}$, where `AllToAll` is the matrix with all entries equal to 1 representing all-to-all collective communications and $\mathcal{E}$ stands for the *unknown* communication pattern introduced by the preconditioner. We find β_0 and β_1 by minimizing the 2-norm of the residual matrix $\mathcal{D} - \mathcal{H}$. In our regression model, we ignored $\mathcal{E}$ since it is not readily quantifiable and we estimated $\beta_0 \approx .3$ and $\beta_1 \approx .7$ with an the r-squared value of almost 90%.

4.4.3 Evaluation Procedure

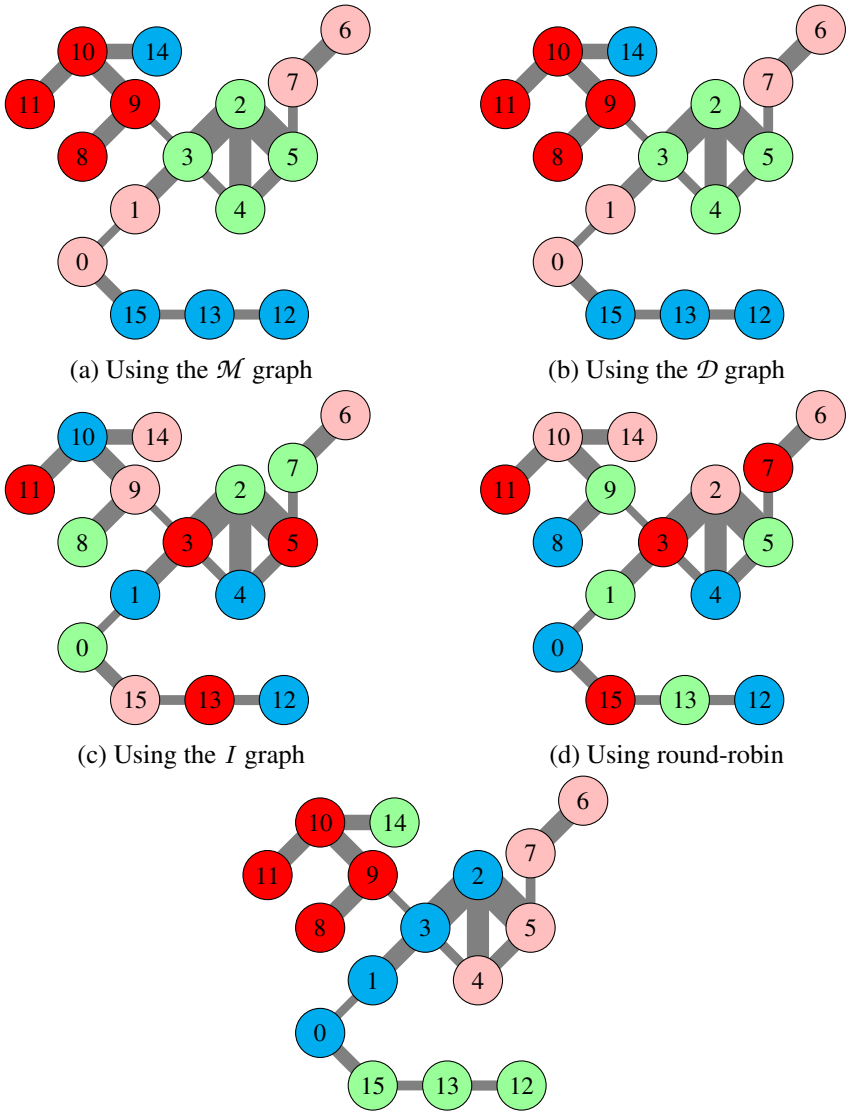
To determine the performance of the different process placement scenarios, we executed all *simulation configurations*, i.e., three mesh resolutions, from 8 to 128 MPI processes, for three

target architectures, using five different placement strategies (total 45 benchmark tests). The first three allocation techniques were formed using the information in $\mathcal{D}$, $\mathcal{M}$, and an additional *inverted* communication graph I defined as $I_{ij} := M - \mathcal{M}_{ij}$, where M is the maximal entry in $\mathcal{M}$. We partitioned these graphs using the `gpart` tool from the Scotch 6.0 software package that implements graph k-way partitioning heuristics [134]. As a result, we obtained three clusterings $\mathcal{C}_{\mathcal{D}}$, $\mathcal{C}_{\mathcal{M}}$, and $\mathcal{C}_I$, such that each cluster included the same number of parts equal to the number of processing units available in a single node of the target machine (4 for `puma` and `lab`; 16 for `amazon`) and the graph cut-sets were minimized. $\mathcal{C}_{\mathcal{D}}$ gave us the process placement optimized to the actual communication statistics. $\mathcal{C}_{\mathcal{M}}$ was the mapping optimized with respect to the partition connectivity graph $\mathcal{M}$ that is a by-product of the partitioning algorithm and does not require any additional work. However, given the strong correlation between $\mathcal{D}$ and $\mathcal{M}$, we expected similar results in these two cases. Finally, using $\mathcal{C}_I$, we expected to receive the worst placement choice.

The second group of allocation strategies were methods commonly used to execute parallel applications with the OpenMPI: *by-node round-robin* and *by-cores* placements. The first allocation strategy places "processes one per node, cycling by node in a round-robin fashion" [131], while the second uses all CPU cores on one node before moving to the next node. These schemata were the control tests to compare our designed placements.

In order to apply the generated mapping, first we prepared the `hostfile` file listing the hosts in the desired order (repeating them on the list as needed). Next, to execute our tests we ran the command: `mpirun -mca rmaps seq -bind-to-core -hostfile hostfile -report-bindings -np N app`. These options ensured that OpenMPI placed the processes according to the order specified in `hostfile`. In particular, the last option is applied to crosscheck the process placement. In each test, we neither oversubscribed nor undersubscribed the hosts—we used all cores available on the computing nodes. Also, we did not perform any tests on single nodes, even if a node could handle the computation alone, as we were interested only in cases using inter-node communication.

To generate the ordering of the host file entries, we resorted to `bash` scripts to provide portability across different targets. Figure 4.5 depicts all five partitioning modes (the first two partitionings are identical) for the *coarse* mesh divided into four parts.



(a) Using the $\mathcal{M}$ graph

(b) Using the $\mathcal{D}$ graph

(c) Using the I graph

(d) Using round-robin

(e) Using per-core

Figure 4.5: Different mappings of the *coarse* mesh for four 4-way nodes. Each color represents a part for a single host. Edge thickness represents the number of common vertices between parts. In this case, the partitionings using $\mathcal{M}$ and $\mathcal{D}$ are the same.

4.5 Results and Discussion

We launched the same hemodynamic simulation using all execution configurations (i.e., target: `puma`, `lab`, `amazon`; mesh resolutions: *coarse*, *medium*, *fine*; number of MPI processes: 2^n for $n = 1..7$) applying five process placement strategies: (1) *data*: based on the communication graph $\mathcal{D}$, (2) *part*: based on the partitioning graph $\mathcal{M}$, (3) *invert*: based on the inverted communication graph I, (4) *pcore*: per-core, and (5) *rr*: by-node round-robin. All tests were repeated twice and the data were averaged. Figure 4.6 shows the execution times for different MPI process placements for all configurations *relative to the maximal execution time* for that configuration (i.e., a chart bar having execution time 1 represents the least effective process placement in the configuration).

As expected, the presented charts demonstrate that deliberate MPI process placement significantly influences the overall performance of the application. In specific situations, the worst mapping in the configuration is almost one order of magnitude slower than the fastest. Comparing the placement strategies, the *pcore* mapping is almost universally a good choice and this type of allocation cannot be often improved by *data* and *part*. The success of *pcore* stems from the algorithm applied in our offline partitioner (cf. subsection 4.3.2): the used ParMETIS routine generates partitions in such way that continuous ranges of length 2^i (for any i) correspond to contiguous parts. Moreover, ParMETIS minimizes the edge-cut between such 2^i parts. Consequently, such process mapping is suitable for 2^i-way multiprocessing nodes by assigning the neighboring, most communication intensive parts within (single) nodes.

Our *data* and *part* placements in a few instances slightly outperform *pcore*. This is true in particular for more demanding cases where the *fine* mesh is processed by 64 or 128 processes. Comparing *data* with *part* alone, the second method seems to be more effective. This may be due to the limited information included in the $\mathcal{D}$ graph which is used to generate *data*—we focused entirely on the total communication volumes and did not consider the number of messages exchanged by the process which may play a key role on high-latency networks. On the contrary, *part* consists of more generic data, i.e., shared entities between partitions, that indirectly represent the entire communication characteristic. Another valuable facet of *part* is that this partitioning is obtained from the $\mathcal{M}$ graph which is provided directly from the partitioner and there is no need to pre-execute the simulation in order to obtain communication statistics.

The worst placements are *rr* and *invert*. In light of the *pcore* explanation, the round-robin allocation breaks locality, mapping consecutive partitions to different machines. The more hosts involved in the execution, the less local are the parts processed by a single node. This is confirmed in observations for `lab`. By construction, the *invert* mapping is the worst placement as we intentionally forced closely collaborating processes to be on the separate hosts.

As we increase the mesh resolution, the computational complexity increases in terms of how many mathematical operations are performed by each process on locally available data. Problem partitioning distributes the computation kernels although the communication requirements are increased. Consequently, to effectively use a chosen target for a parallel application, the hosts' computational power and the network utilization must be balanced. This effect is visible when particular targets are considered, as discussed below.

`puma` has relatively slow CPU's while the network is a high performance-class interconnect. Due to the slower computation phase, the network may deliver needed data promptly in any partitioning scheme for a smaller number of MPI processes (better placements improve the execution time only by 10–20%). However, when more parallel processes are used and the work per unit decreases (relative to the communication traffic which intensifies), the effect of good placement selection is evident (60–80% faster).

`lab` peer-to-peer desktop computers possess fast CPU's and the standard intranet communication fabric. On this platform, for the *coarse* mesh the computation effort for any number of processes is low and processing phases have to wait for more data to continue computation. In other words, the more the MPI processes, the less improvement can be expected (less computation per node with more data to exchange). For more computational demanding cases the efficiency of proper process placement increases with the number of processes—dividing the mathematical kernel into smaller parts requires faster data delivery from adjacent computing units.

On `amazon`, overall performance of a parallel application is greatly affected by the network quality, especially the extremely high latency values [64]. For this reason, there is no reasonable improvement when most process placements are applied. However, the *pcore* placement is especially effective in this case. We attribute this effect to the internal architecture of the Amazon hosts used in our tests—each computer consists of two octa-core processors and mapping MPI processes in the *pcore* way takes advantage of processor affinities while our *data* and *part*

mesh type	target	# of procs	# of nodes	mapping	time [s.]
coarse	puma	16	4	*pcore*	85.7
medium	amazon	32	2	*part*	399.5
fine	lab	32	8	*part*	664.1

Table 4.2: Best time-to-completion for three problem sizes

placements neglects this capability.

Ultimately, for end-users, the most important characteristic of the application is how fast the given problem can be solved, which target should be chosen, and how the execution environment should be configured. In other words, the conclusions of our analysis should guide users in their specific choice of a platform—and a process placement strategy—for a given problem size. For the application under study, such a guide might be given by the data shown in Table 4.2, which also lists the minimal execution times for each resolutions of the input mesh.

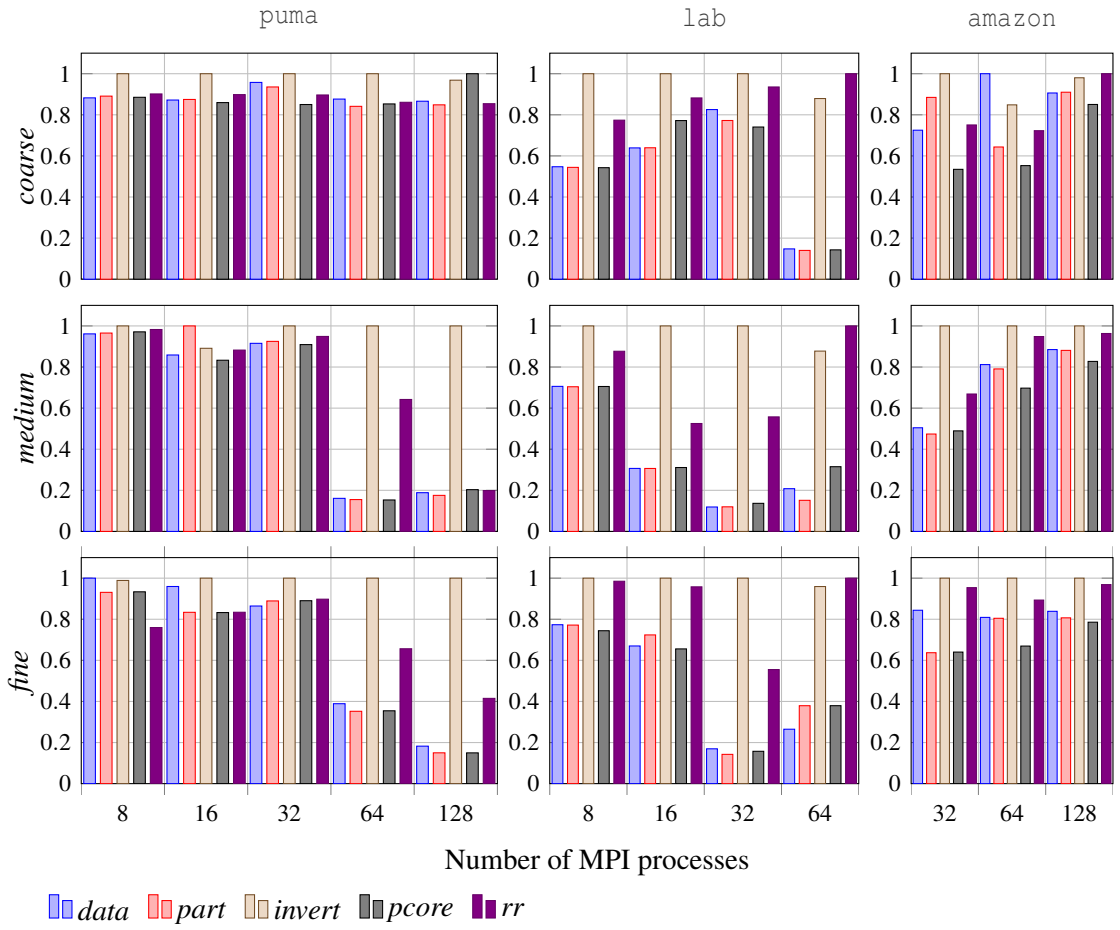


Figure 4.6: Relative execution times for all simulation configurations. Each value represents the speedup of the mappings in relation to the less efficient mapping for the configuration.

4.6 Conclusions

In this work we analyzed performance variations caused by interconnection communication channel heterogeneity and process placement strategies for a parallel CFD application based on the finite element library LifeV. The communication profile for this parallel application depends greatly on the partitioning of the mesh representing the physical geometry of the input. Such communication imbalance invites exploration of the possible mappings of the parallel tasks onto diversely performing networks of processors. As parallel target platforms universally present heterogeneous inter-process communication capabilities when nodes are multicore, performance advantages are possible through process placement that exploit this knowledge. However, such an approach requires a case-by-case process mapping. As benchmark platforms, we selected three parallel targets that differ in many aspects: network fabric (1GbE, 10GbE, IB), the number and performance of processing units per node (2x2, 4, 2x8) as well as the OS and machine purpose (task batch oriented, interactive user oriented). A second aspect of heterogeneity comes from the input problem itself: we simulated the same blood vessel at three different levels of detail that vary computational demands from trivial to intensive. More important to our exercise, the different cases exhibit significant variation in volume and frequency of communication between partitions.

We studied five process placement strategies: three of them use problem-related information and the others are typical OpenMPI process allocations. We found that the standard *pcore* placement mapping is well-suited for processing our CFD application implemented with the LifeV library. The reason for this behavior has its source in the implementation of the ParMETIS partitioning used by the application. ParMETIS uses recursive bisection, which matches the common 2^i-way multiprocessing architecture of contemporary computers. However, we showed that this allocation may be improved by our *part* placement, especially for larger numbers of hosts performing the computation. Moreover, to design such enhanced placement no extra computations, e.g., evaluating communication statistics, are needed—the by-product information regarding the pairwise shared mesh entities from the partitioner phase can be utilized instead. As the result, we showed that it is possible to adapt performance of parallel MPI applications by communication-aware placement of their MPI processes if the computation structure, input geometry as well as target architecture and network wiring is known and it may be done automatically by `ADAPT`.

4.7 Contribution

The time-to-completion is an important metric for scientific software and for this reason SaE applications are parallelized. As a result, we are interested in maintaining perfect weak scalability [65] which mostly rely on the quality of the communication infrastructure (only embarrassingly parallel problems are immune from the communication substrate). If processing elements were fully connected and communication channels were homogeneous, mapping of application processes onto physical nodes becomes insignificant. In reality, all parallel computational targets encompass multi-level communication channels that range from high speed, low latency, core-to-core, in the same affinity zone data movements to slow cluster inter-domain or even Internet networks. This demands careful parallel software–hardware mapping to balance use of heterogeneous communication links. This is especially important if the parallel application does not have well-defined communication patterns or these patterns depend on input.

Typically, users trust that high-performance machines, such as supercomputers, execute their codes in an "optimal" manner; in fact, batch-queuing systems, which allocate the computational resources for jobs, do not tune node allocations based on the logical communication pattern and the execution is suboptimal. Our solution aims at improving performance for parallel applications with unstructured communication patterns executed on modern computational resources equipped with heterogeneous communication infrastructures. We approach this by evaluating situation-specific matching between the communicating processes and the physical computing elements, typically CPU cores. In our idea, we represent the logical and physical communication topologies as weighted graphs (the edges represent communication links with their capacities) and perform graph partitioning/matching to minimize data movement through long-distance connections, while usage of high-speed links is maximized. Application communication graphs may be estimated based on metadata describing the input problem, such as physical geometry of the simulated object, or found based on benchmark runs. Concerning physical communication graphs, the actual physical topology may be unknown until just before the execution, as in the case of high-performance platforms managed by job schedulers. In this case, the matching must occur in a batch script file.

As our method improves the time-to-competition characteristic, it increases the throughput and productivity of SaE resources. Thanks to that, scientists may verify their theories faster and cheaper

as resources are utilize better. Automation and focusing on runtime may prevent hard coding of the communication optimization in applications which translates to (1) more time for developers to focus on functionalities and (2) higher software reliability as less unintentionally introduced errors are made in the code.

Chapter 5

Cost Adaptation for Time Critical Application Execution

5.1 Background

IaaS offerings, such as Amazon's Elastic Compute Cloud (Amazon EC2), provide access to fully configurable computing resources via virtualization. It is therefore possible, in theory, to construct high performance clusters on IaaS cloud platforms for scientific applications, instead of local acquisitions or even grids. In practice however, two issues temper this abstract possibility: (1) the general purpose nature of interconnection networks on IaaS clouds often degrades communication-sensitive parallel processing and scalability (as it was shown in Chapter 4) and (2) cost and turnaround-time tradeoffs may make IaaS clouds less attractive, especially in academic and research settings where on-premise resources may be easier to acquirer, access, or precondition (as it is shown in Chapter 3) as compared to cash resources.

Normally, applications in the HPC domain are characterized by compute- and/or data-intensive codes that are parallelized explicitly, commonly based on the message passing programming model. The (computer science) emphasis during the phases of development and enhancement of these codes is on *performance*, commonly characterized by a single time-to-completion metric, parameterized along two dimensions defining scalability: *problem size* and *number of processing elements* used. However, during production runs by application scientists, turnaround time and cost *may* become

higher priorities. In other words, a scientist may prefer to run a simulation for an hour on a slow, immediately-accessible platform (e.g. local or cloud) instead of on a supercomputer (e.g. on a grid) that takes only minutes for running the same simulation but has an *overnight turnaround* batch queue. With the advent of IaaS cloud computing, this scenario manifests with an additional dimension—that of configuring a cloud cluster where, in theory, performance and turnaround improve with increasing payment [66].

In this study, we analyze our experiences with executing a FEM code from an academic research project with potential use in medical diagnostic code on platforms that have differing cost and turnaround characteristics. Our simulation is representative of scientific computing in biomechanics and numerically simulates blood flow in a cerebral aneurysm. We present measurements of execution times and their pertinent use costs that are actual payments in the case of clouds or estimated cost in case of other environments. These estimates are based on normal parameters including capital cost, operation cost, etc. amortized over normal life cycles. We then propose a model for the subject-specific *utility function* of the computation profiled for different user scenarios.

5.2 Related Work

User-centric performance analysis has recently been applied to research on HPC and Grid scheduling strategies. The value that users associate to a completed job is modeled as a utility function with a generally non-trivial dependence on time [67]. In other words, the importance of a job to a user can be seen as a function of time, combining an index for the *importance of the results* and the user *sensitivity to delay*. The design of proper utility functions has been object of study and it has been shown that a proper job scheduling strategy can significantly increase the performance of HPC systems, measured as the aggregate utility of their users [68, 69]. Several works in the literature discuss an extension to this scenario in which heterogeneous resources can be discovered and assembled from an arbitrary set of providers. In this case, the utility for the user may be defined based on a more detailed analysis of user-specific requirements. For instance, requirements may include the features of the physical resources (memory, processor speed, presence of GPU), presence of installed software, or availability of specific services. It is then possible to discriminate between resource providers based on their ability to satisfy the requirements, in full or

in part (*partial utility*) [70]. The evaluation of the utility function can be done at runtime, to decide whether or not to dynamically re-distribute resources to obtain an optimal "quality of execution," i.e., an optimal trade-off between resource savings and performance degradation [71].

In our approach, the platform options available are considered as interchangeable—after an automatic, `ADAPT`-based soft-conditioning process. We assume that each platform is provisioned with an environment adequate to sustain the user's task. We then identify a basic set of user requirements (minimal cost and minimal execution time of the task) to define a user-centric ranking of the tested architectures.

5.3 Cross-Platform Cost and Utility Comparison

In this work, we compared five platforms in three categories representing typical resources to which academic users have access. Our "base" platforms were `puma` and `ellipse`. The third platform was a HPC supercomputer (`lonestar`) made available to the U.S. research community by Texas Advanced Computing Center, representing "grid" resources available to scientists (XSEDE [135]). Two IaaS offerings completed the suite. The first was `rockhopper` [136] offered as a part of Penguin's On-Demand HPC Cloud Service [137] and the second was an IaaS cloud provided by Amazon's Elastic Compute Cloud (EC2) service (referred to as `ec2` in the following). The short characteristic along the cost and utility metrics of those platforms (refer to Chapter 3 for additional descriptions):

`puma` In-house computing cluster with 32 four-core nodes. For comparison with other architectures, we estimated the cost of our department cluster, based on its initial price and operating expenses, at 2.3¢ per core-hour.

`ellipse` This university cluster consists of 256 four-core nodes but does not support execution of parallel Open MPI jobs exceeding 128 processes. All users of `ellipse` pay a flat rate of 3¢ per core-hour (the price changed in comparison to the previous experiment).

`lonestar` This Linux cluster consists of 1,888 12-core nodes. The available queues accept jobs with a maximum wall time of 24 hours on at most 4104 cores. Access to this facility is granted to off-site users upon request of allocation to the National Science Foundation XSEDE project. User accounts receive a certain amount of Service Units (SU) corresponding to core-hours. The

equivalent value in dollars of an SU on `lonestar`, based on an estimate of the acquisition cost and the project cost (personnel, power, cooling, etc.) is 7¢ per core-hour.

`rockhopper` This commercial offering permits a maximum of 256 cores/processes and the vendor charges a fee for using its cluster—we were charged 10¢ per core-hour including an academic discount but excluding marginal expenses for storage use.

`ec2` We picked the most powerful instances `cc2.8xlarge` from *Cluster Compute* and exclusively used `cc2.8xlarge` instances from the `us-east-1d` Amazon data-center zone. In order to provide the execution platform for our application, we launched a certain number of these instances in a single placement group and configured them to achieve an MPI parallel execution environment. During our exercise, the regular price of a single `cc2.8xlarge` instance (16-core, 60GB RAM) was $2.4 per hour while the average spot request price calculated for all simulation runs we launched was 36.35¢ per hour (or about 2.27¢ per core-hour).

5.4 Metrics

Our goal is to characterize differences in effectiveness of the different platforms in terms of cost and utility. In order to do so, we report and discuss one technical measure, i.e., time to completion.

Time to completion is the wall clock time from program launch to final exit. In the mainstream HPC community, in which it is a primary focus, it is not common to include the queue waiting time. In terms of utility in the sense adopted in this example, the queue time is certainly important but it is highly variable and, in fact, our platforms presented few queue delays compared to the execution time of the application. We decided to exclude the queue time in our analysis for the sake of simplicity and uniformity, especially since "on-demand" IaaS clouds are characterized practically by the zero waiting time.

Cost per simulation is the overall cost for the execution of the job depending mainly on the unit cost of the hardware (cost per core-hour), its pricing policy (by core or by node, by hour or prorated), and on the execution wall-clock time. Other factors, that we consider negligible relative to the former (for our application), are the size of occupied storage and/or volume of data staged in

and out. As mentioned above, we experienced the following costs (presented here as a price proxy estimated for a single computational process):

target	core-hour
`puma`	2.3¢
`ellipse`	3¢
`lonestar`	7¢
`rockhopper`	10¢
`ec2`	36.35¢ for a 16-core EC2 instance per hour. Since Amazon charges the users for the entire machine, cost effectiveness decreases if not all cores are utilized.

The Utility function expresses the job's value to a user, as a function of time. This has a user-specific, complex dependency on several parameters, including expenditures, time to completion, and significance of the task. Following [69, 72], we consider a linear utility function with customizable maximum (starting) value and slope as in Figure 5.1. U_{max} is a measure of the *importance* of the job to the user and we assume that we can give it a monetary value as the price that the user would be willing to pay for the simulation. T^* is the expected completion time, which can be estimated in several ways; we use a simple averaging method based on the performance on the available platforms. T_0 is the user-defined time at which the utility is zero while the distance $(T_0 - T^*)$ is a measure of the user's *delay tolerance* and can be measured as a multiple of the expected completion time T^*.

With this formulation, we assume that there is no loss of value during the expected duration of the job (when $t \leq T^*$). An extension of the model could take into account the decrease in the utility function during runtime, reflecting the fact that faster runtime is valuable to users [67].

5.5 Experimental Results

Our experiments on different architectures yielded interesting results. This discussion centers on cost and utility; detailed analyses of communication costs, software issues, and adaptability to different platforms are in [8]. Here, we only present overall runtimes, and, subsequently, a cost-utility analysis.

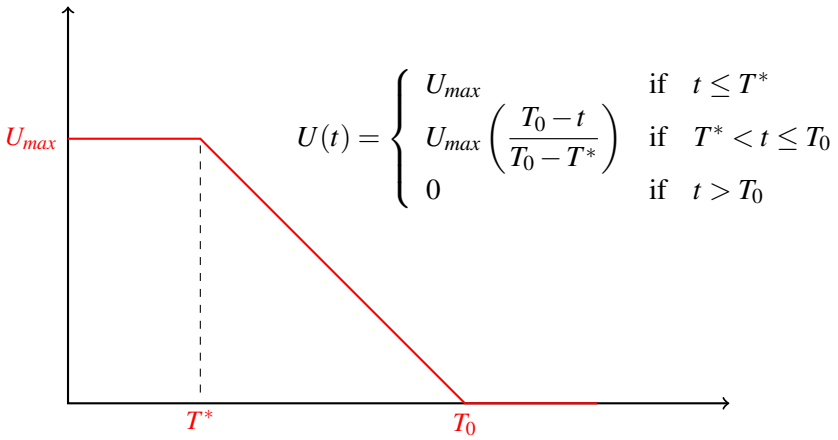


Figure 5.1: The considered utility function

5.5.1 Performance, Scaling, and Time to Completion

In our study, we tested the selected platforms executing a fixed-size simulation (over 3.1M unknowns) with varying numbers of processors, viz. a strong scalability benchmark. All tested platforms permitted the reservation of computing resources by specifying the number of processes (or slots) used by the parallel job. However, in the case of the Amazon EC2 cloud, we needed to set the execution policy: we assumed that each `ec2` instance can host a maximum of 16 processes (as they have 16 physical cores) and we decided to map MPI processes onto the physical nodes in round-robin fashion. As Amazon charges users on the basis of running instances, we decided to optimize the cost of the benchmark by testing small assemblies of `ec2` first, and then to increase the number of nodes in the assembly by powers of 2. For this reason, we present several configurations of cloud instances; we label such separate assemblies as `ec2`-*i*, where *i* is the number of nodes.

The application repeats the same set of operations in each simulated time frame (in our case corresponding to 0.01s intervals). For each considered hardware platform, the time required to compute a single frame was observed to be constant during the course of the simulation. We, therefore, use the average computing time for a single frame as a proxy for the performance of the hardware resource. This facilitates a side-by-side comparison of all platforms, including cases in which the simulation could not be completed due to cluster usage policies (e.g., `ellipse` limits job execution time to 12 hours so for jobs that spanned small numbers of cores only a fraction of the entire simulation could be completed).

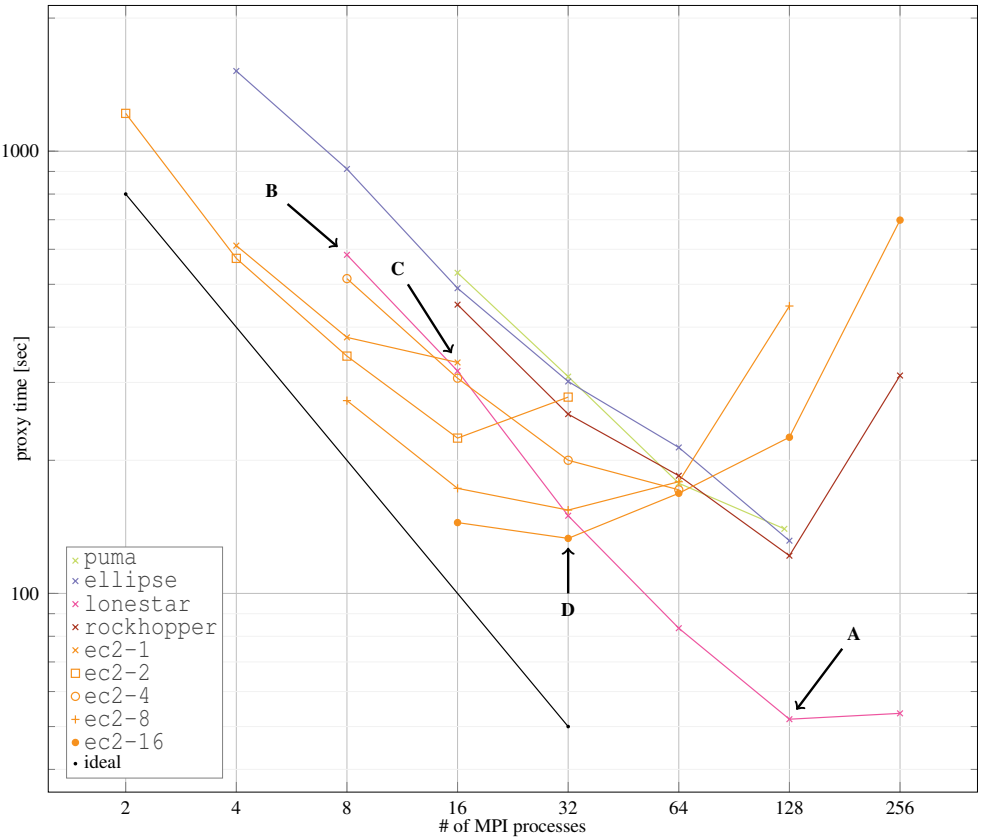


Figure 5.2: The average computation time per simulated time step (proxy) for the benchmarked architectures

Figure 5.2 shows a performance comparison of the different platforms as a function of the number of computing cores. On-premise resources (`puma`, `ellipse`), the HPC cluster (`lonestar`), and `rockhopper` achieve good strong scaling up to 128 computing cores while they show a significant decrease in performance for larger numbers of cores. In particular, Point **A** in the figure corresponds to the fastest execution case that is running the simulation with 128 computing cores on `lonestar`. If this metric is used to represent utility or value to the user, it is clear that when using more than 32 cores, `lonestar` is the best platform.

`ec2` resources scale less well. `ec2-1` achieves good scaling only in the range 4-8 cores, `ec2-2` up to 16 cores, `ec2-4` up to 32, `ec2-8` up to 16 cores while `ec2-16` does not achieve strong scaling in any range. Point **C** in Figure 5.2 corresponds to the case in which the simulation used 16 computing cores on a single `ec2` instance. It is worth noting that the time to completion in this case matches the time to completion obtained using 16 computing cores on `lonestar`. Most significantly, the time to completion required by `ec2-1` when using 8 computing cores is lower than the time required by

lonestar with the same number of computing cores. This highlights one of the main advantages of IaaS clouds; i.e., the availability of powerful hardware configurations (both in terms of memory and CPU clock speed) that can match and outperform the computing nodes provided by typical grid resources. This finding is in agreement with previous reports. A study [73] pointed out that when an EC2 user reserves an entire computing node (this happens in our case using cc2.8xlarge instances) the impact of virtualization is negligible since processor cores are not shared among users (cf. [74, 75]). This results in performance comparable to "bare metal" hardware. As predicted in [76], this is a significant advantage of *Cluster Compute* instances over former offerings by EC2 that suffered from performance degradation due to concurrency of multiple users or applications using the same processor. On the other hand, the performance of ec2 platforms seems to be sensitive to overload of the instances, as shown by the poor strong scaling achieved by ec2-1 when all of the 16 available computing cores are used for the execution of the simulation.

Point **D** corresponds to the fastest execution on ec2 resources, that is running the simulation with 32 computing cores using ec2-16. In this case, we launched 16 EC2 instances and allocated 2 computing cores on each instance in round-robin fashion. The loss of performance of ec2-16 as the number of cores per instance increases suggests that when requiring a relatively large number of instances, the physical connectivity of the nodes may become an issue and the timings seem to be dominated by communication overheads. The severe impact of network latency and bandwidth on EC2 performance is a known issue, especially for large assemblies of instances [74, 75, 76]. In terms of utility, therefore, individual EC2 nodes offer high performance but when communication across nodes or racks is involved, this platform is less attractive.

Based on the metric *time-to-completion*, we can rank the different resources—Table 5.1 shows the wall clock times for the fastest run on each platform. lonestar is by far the fastest resource while ec2 is generally the slowest. However, one of the solutions provided by ec2 (namely ec2-16) matches the performance of on-premise resources (ellipse and puma) using a significantly smaller amount of computing cores. This result further demonstrates one of the strengths of contemporary on-demand resources as compared to on-premise resources, i.e., ec2 can count on cutting edge computational hardware configurations. The rockhopper target performs best among the tested on-demand resources and better than the on-premise resources; However, it is slower than the cluster.

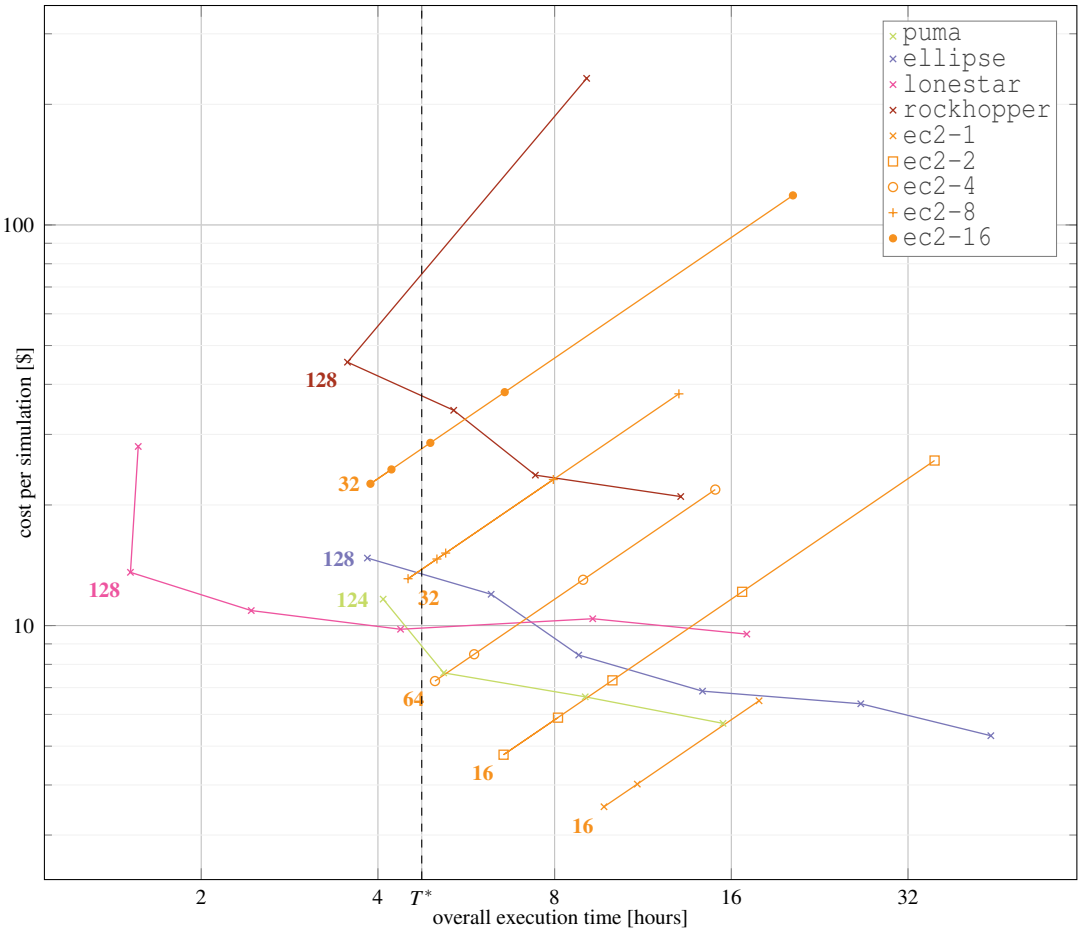


Figure 5.3: Relation between cost and simulation time. The labels indicate the number of MPI processes in the fastest run for that platform. $T^* = 4h\ 44m$ corresponds to the average value among the fastest runs.

Utility function. Ideally, users desire to minimize both simulation cost and time to completion but these objectives compete with each other. Our tests confirm this: the cheapest resource, namely `ec2-1`, was also the slowest. In Figure 5.3, we present how the cost per simulation relates to the time to completion for different architectures. The closest points of the graphs to the origin of the axes represent execution cases that minimize both metrics. Clearly, the decision on which architecture to prefer cannot be made based on a single attribute. The general trend of these characteristics shows an increase in the cost per simulation with decreasing time to completion. A remarkable exception is `ec2`, for which cost increases with time to completion. In fact, on this platform slower executions achieved with few computing cores are actually more expensive due to the policy requiring the reservation of 16-core instances.

rank	time to completion [s]	target	# of MPI proc.
1	1h 31m	lonestar	128
2	3h 33m	rockhopper	128
3	3h 50m	ellipse	128
4	3h 53m	ec2-16	32
5	4h 05m	puma	124
6	4h 30m	ec2-8	32
7	5h 00m	ec2-4	64
8	6h 33m	ec2-2	16
9	9h 43m	ec2-1	16

Table 5.1: Performance based on the *time-to-completion*

To define a ranking of the tested platforms based on a user-centric performance analysis, we evaluate the utility function defined in section 5.4. We consider three user profiles: (1) high job priority/little user's delay tolerance, (2) average job priority/average user's delay tolerance, and (3) low job priority/large user's delay tolerance.

Referring for the sake of example to the results of our benchmark, we consider the cost of the cheapest use case for each platform (cf. Figure 5.3). This gives us a range from $3.53 (the cost of the cheapest execution on ec2) to $22.59 (the cost of the cheapest execution on rockhopper). We assume that the value of a simulation to the user (i.e., the cost the user would be willing to pay) is within this range. More precisely, we assume that a job with low priority has a value to the user equal to the average cost of the cheapest use case over the tested architectures, i.e., $10.31. We assign double this value to a high priority job ($20.62) while an average priority job will have an intermediate value between the previous two ($15.465). We further assume that for all the user profiles the expected time to completion T^* is the average value of the times measured on the different architectures (cf. table 5.1), i.e., $T^* = 4\text{h } 44\text{m}$. A user with an average delay tolerance is represented by a utility function that remains non-negative for a runtime up to twice the expected value (i.e., $T_0 = 2T^*$). A user with large delay tolerance accepts twice as much delay ($T_0 = 3T^*$) while a user with small delay tolerance accepts half as much (i.e., $T_0 = 1.5T^*$).

We plot in Figure 5.4 the user-specific utility functions together with the graphs shown in Figure 5.3. As discussed in previous sections, each platform was tested in several use cases (varying the number of computing cores); a use case is considered *useful* to the user if it is represented by a point on the cost/time plot located below the graph of the user's utility function. For the sake of

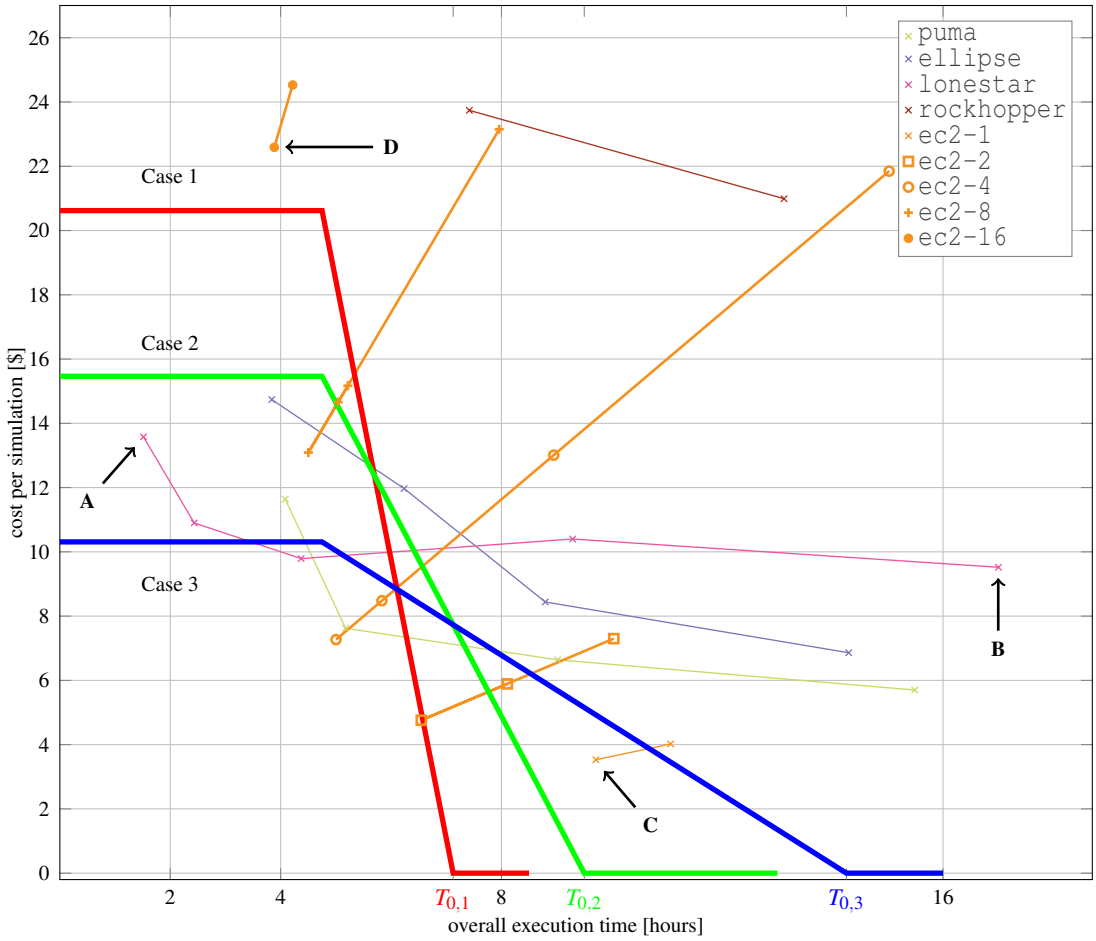


Figure 5.4: Evaluating the cost/time characteristics of the different platforms against the user-specific utility function. $T_{0,1}$, $T_{0,2}$, and $T_{0,3}$ are the times at which the utility function is zero for user profiles 1, 2, and 3, respectively.

example, we reported on the plot the points corresponding to the cases discussed in detail in the previous sections. Point **A** corresponds to the fastest execution of the simulation in our experiment, obtained when using 128 cores on `lonestar`. This case is *useful* to user profiles 1 and 2, for which the *importance* of the simulation is greater than the actual cost. User profile 3 would not consider this case *useful* due to its high cost.

Despite the cost being relatively lower, the use case of `lonestar` represented by point **B** (8 computing cores) is not *useful* for any user profile, because for all of the profiles the time to completion of the simulation exceeds the time T_0 for which the utility function is zero. The use case of `ec2-1` corresponding to the cheapest execution in our experiment (16 computing cores) is represented by point **C**; because of the long time to completion, this use case is only *useful* to user

profile 3. The fastest execution achieved on ec2 resources (2 computing cores on each instance of ec2-16) is represented by point **D**. This use case has a cost exceeding the maximum value of the utility function for all the user profiles, so it is useless.

In our experiment, a variety of platforms can meet the requirements of user profile 1. Fast and expensive architectures (e.g., lonestar) can be chosen in alternative to slower and cheaper ones (e.g., ec2). However, because of a small delay tolerance, a cheap option (ec2-2) has to be ruled out, being penalized by high execution times. The second user profile has the largest pool of *useful* choices, including the cheaper (and slower) ec2-2. For user profile 3, because of the low priority assigned by the user to the job, most of the fastest options (lonestar, ellipse) have to be discarded. On the other hand, the on-premise cluster puma and some of Amazon's instances (most significantly the very cheap ec2-1) can meet the user's requests.

According to this model, one of the on-demand IaaS cloud resources (rockhopper) is *not useful* to any of the considered user profiles. The other IaaS provider (Amazon), however, has a variety of offerings and pricing schemes—Amazon's diverse offering allows its service to be competitive for a wide range of user profiles, being able to provide reasonably small execution times (ec2-8) or extremely cheap solutions (ec2-1). On-premise resources (puma, ellipse) do not perform well compared to HPC machines (lonestar), being significantly slower, and in most cases they are also outperformed by cheaper on-demand resources (ec2). As a result, they are competitive only in specific execution cases (i.e., with the proper choice of the number of computing cores). Finally, lonestar is a very strong competitor in the first two user scenarios (average to high job priority) while its performance is matched and outperformed both by on-demand and on-premise resources in the third scenario (low job priority and high delay tolerance).

Notably, the on-demand cutting edge offering by Amazon EC2 has the advantage of availability. In fact, our analysis does not consider any queue waiting times that may diminish the attractiveness of shorter execution time on grid resources. This feature would make the IaaS choice even more convenient. Moreover, the cost per simulation on the resources offered by Amazon can be optimized with a proper scheduling policy that takes into account the specific pricing policy of Amazon (per-node rather than per-core). Furthermore, to minimize cost, it is possible to select cheaper instances such as cc2.4xlarge.

5.6 Summary

We have presented experiences and utility analyses based on our experiences with production CFD code on five different target platforms in three typical settings: on-premise resources, grids, and IaaS clouds. Comparing execution time and cost of the application on on-premise and on-demand targets, we found some evidence to support the claim that IaaS resources may be utilized for scientific CFD simulations possibly at lower cost than incurred locally. In particular, our test with Amazon's spot-request feature coupled with availability of cutting edge resources (16-core nodes, 60GB RAM) suggests that small on-demand assemblies may be a viable alternative to local clusters. It is not without significance that IaaS's provide resources immediately, while local and grid resources are often subject to possibly long queue wait times—an aspect that might offset any additional expense. Furthermore, while a modern local computing cluster with an efficient interconnection network will outperform an on-demand assembly (which is highly vulnerable to network performance) the cloud solution might be useful when cost needs to be minimized.

In the context of our project, if the utility function can be built-in the `ADAPT` execution environment, the users may parameterize a specific run of an application giving the maximal cost and time-to-completion as the execution input. `ADAPT`, based on the cost and time models for available resources and characteristic of the application, may adaptively select the cheaper target that guarantee completion of the users' request.

Chapter 6

Adaptive Deployment Automation

6.1 Background

Software modeling and simulation have became basic tools in Science and Engineering and steadily replace traditional scientific "hardware." Frequently, software provides only methods to understand observations (for example "big data" science [77]), predict the evolution of studied phenomena [78], or design systems with certain properties [79]. Another reason of increase use of computational methods is to take advantage of well established numerical techniques to improve scientific models that traditionally are not "computational" or make them more tractable [80]. However, cherry-picking of different software tools that represent different computational frameworks, execution paradigms, or runtime systems leads to inherently heterogeneous software mesh-ups. As the result, contemporary SaE software ecosystems require additional attention at various levels such as development, execution, or data processing.

On the other hand, traditionally scientists execute their applications on computational clusters located at datacenters and may rely on expert knowledge if they trouble with software deployment and runtime. However, the advancement in computer science and technology gives more options to host the SaE-class applications outside datacenters and they may be run also on powerful, multi/many-core workstations and computational clouds. Despite unavoidable performance degradation in some applications [64], new platforms attract for various reasons: instant resource availability (no job queues; any number of hosts), cost (no upfront expenses), or greater control (root privileges). In opposite to these benefits, exacerbated target diversity and lack of support harshly

hamper usability of these new targets, especially as SaE applications are usually hard to deploy.

We characterize SaE applications as sets of cooperating programs (usually parallel) with the following attributes: source code availability, nontrivial software dependencies, and target optimization necessities [81]. The cutting-edge nature of these issues impedes using SaE applications beyond well-supported environments, such as developers' or HPC centers' platforms. In the extreme cases, users confine themselves to specific software and hardware solutions to avoid troubles—this reduces productivity, innovation, and may be more expensive. This clearly conflicts with the goal to promote free experimentation with available SaE software tools in science. The users must face challenges related to system software preconditioning and SaE software deployment and execution, which are problematic and time consuming even for highly-trained specialists at HPC centers [82].

Clearly, a more autonomous approach in SaE application deployment and improved model of execution are needed; however, these subjects in the context of scientific software have not attracted sufficient research attention yet and solving software preconditioning issues usually imposes an unnecessary burden in scientific community. This `ADAPT` exploration delivers a description of a deployment and execution toolkit `ADAPT-D` that provides a dynamic and transparent multi-target deployment solution. This design employs dynamic and reusable *deployment recipes* that capture expert knowledge pertaining to solving specific problems during soft conditioning. A proper arrangement of those recipes deploys requested SaE software on selected targets in on-demand and transparent way. Moreover, this tool approaches the deployment in an adaptive manner and reduces the deployment steps to required minimum for a target given as it uses already preinstalled software. Thanks to the fact that we focus on (but do not limit to) the source code-based deployment, which is the norm for SaE-class applications, we may address broader spectrum of computational resources because prebuilt binaries cannot be used everywhere and other resources, notably clusters, do not offer software installation from packages for unprivileged users. We believe that smooth and unsupervised switching between computational providers, even for a single application run, promotes experimentation and evaluation of the SaE software and nurtures the progress in science.

6.2 Related Work and Concepts

Build automation software (e.g., GNU Make [138], CMake [139], Ant [140], SCons [141], Gradle [142]) tools are single project-oriented and often target specific programming languages, such as C/C++, Fortran, or Java. The off-the-shelf build tools have proved its usability for applications that have typical requirements and follow standards common in software engineering. The main disadvantage of common build systems is that they do not handle dependencies and communicate with the users using cryptic build error messages that must be "translated" by operators into additional deployment steps (usually, they pass through messages from deployment tools). Moreover, maintaining a valid deployment configuration from alternative dependencies may be challenging. We propose a more abstract approach based on the concept of *metadeployment* that controls provisioning of software packages. The metadeployment directly uses installation mechanisms as they are given for the interesting software; however, it actively *monitors* the native deployment process and *reacts* on errors. Thanks to that, the metadeployment *dynamically* discovers missing dependencies, *adaptively* conditions them, and retries the deployment. In this sense, our proposition does not introduce yet another replacement for build systems but *coordinates* existing deployment methods.

A few build tools, notably GNU Autoconf [143], support *probing* functionality that checks presence and appropriateness of requested dependencies, such as specific versions, offered computation precision, or available API calls, before the actual build is started. Despite some tools may recognize errors detected (e.g., CMake), in general users must explore the compilation or execution logs of probing tests to search the reason of the fault and understand how to fix it. Our tool goes a step further and not only recognizes the probe/build errors but also fixes them. This process continues automatically until the software compiles successfully. In comparison to traditional methods, our methodology proposes a reverse solution as `ADAPT-D` initiates user commands in unprepared runtime environment and conditions it based on feedback from the system tools. Such the *lazy evaluation* greatly simplifies a design of any probing facility. Moreover, this mechanism is applicable to execution time too where it may monitor an application runtime and react on errors.

Multitenant targets typically offer multiple versions of preinstalled tools and scientific libraries; to navigate among them, users may use Environment Modules [144] that modifies environment

variables enabling requested software packages. In ADAPT-D, the metadeployment algorithm uses *deployment recipes* to *enable* requested software. These recipes capture any actions that users would perform in order to make the needed software available, in particular, the actual deployment or module load steps. ADAPT-D allows defining *alternative* recipes that are *tried* in a strict order that may prioritize environment module manipulation over deployment from source codes.

Package management systems, such as RPM [145], greatly simplify installation of software packages and offer automatic dependency resolution. Such packages include the software payload (either precompiled or as source codes) and a dependency inventory. However, even if SaE applications are distributed as standardized packages, their up-to-date versions are rarely supported by maintainers and, in practice, the software has to be built from the source code. In addition, SaE build systems are often proprietary, which may greatly hamper unification in a chosen package format. Moreover, SaE software often requests specific versions, selective compilation, or patching its dependencies [81] which cause that any SaE application deployment toolkit must support extraordinary SaE application build requirements. For these reasons, even if it is possible to provide a homogeneous build method for an application and all its dependencies as, e.g., RPMs, it is impractical [83] and we do not aim at it. In ADAPT-D, the deployment recipes may also include any package management system command that may rapidly enable requested software but they may be applied only if the users work in the elevated privilege.

EasyBuild [83] facilitates installation of SaE applications by standardization of common deployment steps such as downloading, configuring, or compilation. Deployment actions for software components are defined as Python classes extending the `EasyBlock` class; further declarative specialization sets deployment details such as a compatible toolchain or download URI. In our approach, users save their actions that enable the selected software in a recipe that is later executed during deployment by an user's agent. Customarily, recipes have form of shell scripts and, thanks to that, users can directly save their knowledge for future reference and reuse without mapping it to any imposed scaffolding. From a broader perspective, EasyBuild follows the traditional *bottom-up* approach, while ADAPT-D is dynamic, selective, and employs an optimistic assumption about the environment being conditioned (a *top-down* approach).

The conventional operating system organization helps deployment tools detect software requirements in well-established locations (e.g., `/usr/local`) or in locations pointed by some

environment variables (e.g., `PATH`, `JAVA_HOME`). However, the users are free to use any installation path for software deployed manually which may blind requirements detection mechanisms. For this reason, some deployment tools "register" installed software—EasyBuild extensively uses Environment Modules to deploy and solve dependencies, while CMake collects metainformation about software installation and its configuration. We promote a similar idea and `ADAPT-D` defines a standard, user-space prefix for installations as well as modifies the execution environment to easy detection of the software by standard deployment tools.

Other interesting projects aim at improving portability of software execution between different targets. For example, the CDE project [84] uses `ptrace` to capture and preserve all resources, including dynamic library files, used by an application in runtime. Than, such an assembled package may be conveyed to another machine and executed by CDE, which delivers needed resources from the saved package. Similar philosophy is implemented in PortableApps.com, where the self-sustained packages are provided to execute. However, these methods cannot withstand with heterogeneity typical for SaE applications and targets as well as forbid tuning for architectures that is demanding in high-performance computing. In addition, SaE software might be linked statically with no external runtime dependencies, which surpasses that solution. Another technique may exploit sandboxing, e.g., as implemented in [102], to deliver single application-oriented build and execution environment. Even all the techniques may facilitate a certain facet of deployment, the pure software conditioning is still absent. `ADAPT-D` aims at supporting users with software deployment (focusing on the source code build) and execution automation regardless a target selected and approaches it using alternative recipes created for situation-specific cases.

6.3 Automatic Deployment Description

In this section, we give the description of the `ADAPT-D` design, its sources of inspiration, and used algorithms. The goal is not to invent a more generic deployment method but to provide a metadeployment system that manages and uses already existing deployment mechanisms as they are for various software packages. Figure 6.1 shows the overall deployment situation. An application to execute may require some dependencies that must be met in order to deploy it; conversely, a target planed to be used may offer some deployed already dependencies. In fact, these two sets are of the

same type: they are *resources* (files) staged in the target's file system and we call them *capabilities* to distinguish them from other files. Generally, the capabilities are requested and delivered in packages, e.g., sets of libraries associated with their headers, and therefore it is convenient to operate on *capability sets*.

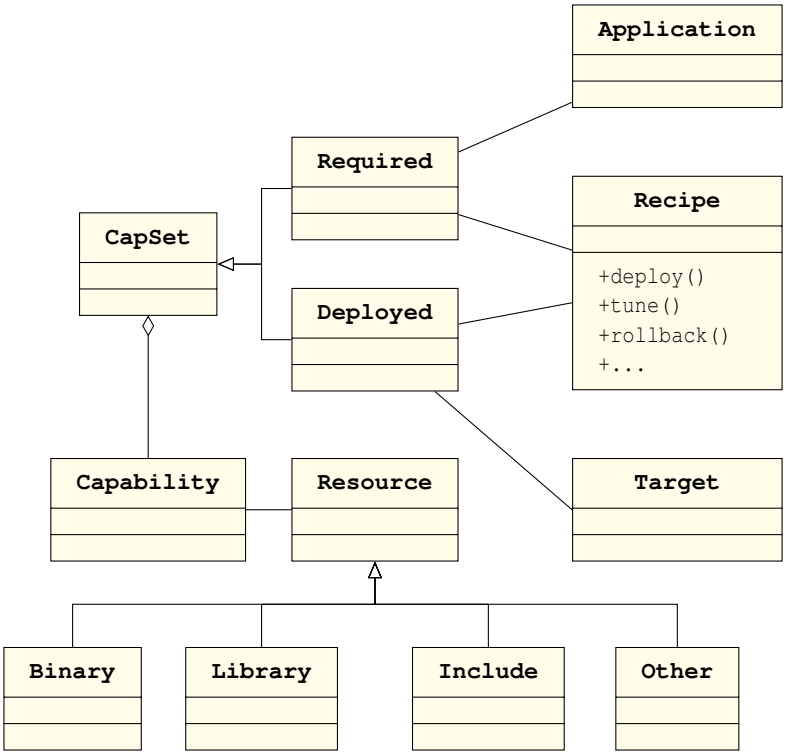


Figure 6.1: The model of deployment in ADAPT-D

If the required capability set is a subset of the provided capability set, the application can be executed instantly; otherwise deployment recipes (deployment actions) must be applied. Note, that these recipes alone act as "applications" and may require some capabilities before they can be used. Therefore, an atomic component for a metadeployment system is a deployment recipe that delivers a capability set.

The main idea is to reproduce users' regular conduct with software deployment. During the software provisioning, the users apply their knowledge and solve issues that occur throughout this process. When the users face errors that require extracurricular experience, they may use question-and-answer web sites to get aid from the community. Usually, a *description* of the problem is sufficient to get a *solution*, apply it, and move forward with the installation (or at least to a next issue).

Similar approach drove the design of this toolkit that manages recurring error–fix phases until it yields requested software installation. The advantage of human operators is that they may make information from `README`-like files usable and can fully precondition the build environment. To mimic this, an automatic deployment tool should *discover* such the dependencies, which is problematic in the view of different programming languages, deployment techniques, etc. We overcome this issue in another way and *induce* the tool to automatically *detect* deployment issues. `ADAPT-D` presumes an *ideal execution environment*, i.e., `ADAPT-D` always tries to execute a command regardless its availability, and analyses the execution status, which provides a chance to detect errors, such as missing requirements, and react on them deploying needed software or fixing its configurations. Figure 6.2 shows schematically this idea. Users must use the `adapt` wrapper in order to apply the enhanced execution of a command. In the iterative manner, `ADAPT-D` gradually softconditions the runtime environment, until all requirements of the command are met. Note that as `ADAPT-D` monitors *execution* of commands, it may detect both pre-execuiton and in-execuition requirements.

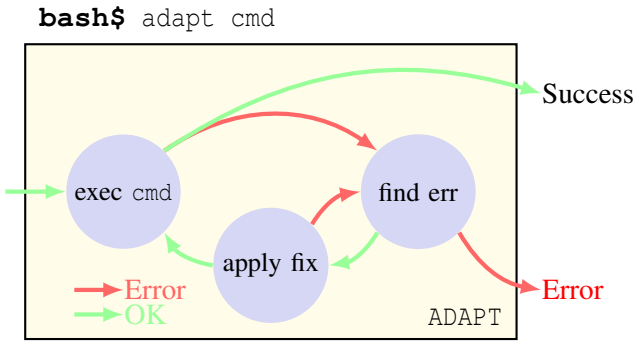


Figure 6.2: The user executes a command in the `ADAPT-D` monitor. If execution fails, `ADAPT-D` detects the error and tries to repair it; than, the execution repeats.

In a typical scenario when usefulness of new software is evaluated, users first deploy the software and than try it in a small scale. As most SaE applications and build tools are command-line, a shell is the best choice environment. For this reason, our toolkit provides a set of command-line tools (currently in `bash`) that support the interactive execution interface. Nevertheless, it does not limit other execution methods, such as a job scheduling, as after an initial deployment phase, the conditioned software is available and ready to use even without the `ADAPT-D` infrastructure. The following subsections describe in greater details the basic elements of the proposal.

6.3.1 Execution Model

Providing the reliable execution mechanism is a focal element of the project and ADAPT-D is designed to execute a requested command in a transparent way. If an application is installed and functional, it is executed directly; otherwise, the tool performs a series of errors' detections and repairs that finally enables the command on the target. As the actual errors depend on capabilities installed on a target given, ADAPT-D approaches the software preconditioning in an selective way providing only necessary deployment steps.

Figure 6.3 shows the algorithm of command execution in ADAPT-D. As the tool may work recursively, adapt maintains the usage context saving a new *context frame* on a *stack* (the buildup of the stack is shown in figure 6.4). Then, the given command is executed by the tool that transparently captures the standard output and error streams along with the exit status and saves them as the frame data. In the next step, ADAPT-D determines if the command was executed successfully; however, it cannot rely barely on the exit status and a more command-specific approach is needed. To assist this process, we implement and use throughout our project some object oriented programming (OOP) concepts (outlined in 6.3.3), which are denoted in the diagrams by the "dot notation." Thus, the obj.check_status method checks the correctness of the execution based on analyses of the data collected in the current obj frame and, optionally, command log files (this method defaults to getting the error code). The success finishes the execution; otherwise, ADAPT-D proceeds to enhance the runtime environment.

The error detection employs the *error evidence* generated by the get_error method that works on data produced by the command execution (this method defaults to the content of the standard error stream). Next, the error evidence is compared against error patterns (regular expressions) delivered by get_ehandlers; each pattern is associated with a solution that is a command to execute. For example, the simplest and most generic error handler is .* (.+): command not found and the associated command is adapt deploy cmd/\L\1, which, for a missing exampleCMD command executed in bash, expands to adapt deploy cmd/examplecmd. This process is performed first on the current context (the top frame) and is continued for the next frames in the stack until a solution is found; thanks to the contextualization, error fixes may be associated with their proper context and applied even if the error is generated by an internal

command in a higher context, as it is explained in figure 6.4. If the fix works, execution of the original command is repeated; otherwise, `ADAPT-D` cannot solve the issue and this command fails.

6.3.2 Deployment Model

The most common reaction on errors caught during `ADAPT-D` execution is the `adapt deploy res` command, which delivers the actual deployment solutions for the `res` resource, i.e., a command, library, or software package; the deployment algorithm is depicted in figure 6.5. The actual deployment steps are defined in *deployment recipes* that are just shell scripts and contain exactly the same commands that the users apply interactively when they are provisioning the requested software. Thanks to that, creating a deployment script is as easy as preserving those commands in an executable file. Usually, there are many ways to provide the same resource, e.g., to enable the LAPACK library, one may use an RPM package, yum/apt-get package management utility, or perform the installation from sources; however, such the variants may not work in all situations. For that reason, we introduce the concept of *alternative* deployment recipes—a single resource may be associated with several deployment recipes that are selected to use by `adapt deploy`. Each recipe should provide a single *configuration*, i.e., the actual installation, for a resource.

The current proposition is that a resource installed by a recipe can have only one active configuration. If a deployment request repeats, it invalidates the active configuration because if another error triggers the same deployment, it proves that previous deployment was ineffective. In that case, `ADAPT-D` chooses a next recipe from the set of alternative recipes and tries it. However, we allow for configuration *tuning*—some software packages may be reconfigured by enabling optional features, e.g., switch on MPI, use of an external LAPACK library, etc. In this case, if newly requested features differs from the previously applied features, then the configuration may be amended by its recipe; otherwise, this request also invalidates the configuration.

`ADAPT-D` tries and uses the deployment recipes in predefined order and when all recipes have failed the deployment fails. Finally, users may also use the `adapt deploy res` command directly to deploy a software of their interests.

6.3.3 Repository and Object Oriented Mapping

ADAPT-D software is associated with the *repository* that delivers deployment recipes for requested resources. As it was mentioned, to facilitate and organize ADAPT-D implementation, we designed a few OOP concepts to use in bash and these features are ingrained in the structure of the repository. ADAPT-D maps a hierarchy of classes into a hierarchy of directories in the repository—the root directory of the repository serves as a root of the inheritance (as, e.g., Object in Java) and its subdirectories are treated as classes; figure 6.6 shows an example UML diagram for a repository. To reference to an object we use relative to repo paths, e.g., cmd/cmake or lib/trilinos; with the cmd prefix applied by default. In a similar way, methods are emulated by bash scripts and attributes are stored in the meta file in each class. To replicate the *virtual calls*, we developed another ADAPT-D tool adapt oop obj method param_list, where obj is the object reference; self can be used instead of obj to point the current frame. An example of use of this mechanism is the get_error() method that delivers the error evidence for adapt. It is defined in repo and, by default, provides the stderr stream; however, if a particular command does not expose its error in stdout, this method must be overwritten, as it is for bjam from boost.

To get and set values of fields we use another ADAPT-D tools: adapt query obj var and adapt register obj var value. The query infrastructure retrieves also ADAPT-D configurations, e.g., make -j $(adapt query sys/cpunum) will use all target cores during build. As the idea of ADAPT-D supports the lazy evaluation and initialization, we encourage users to use information about requirements without checking their validity. For example, to retrieve information about the path to libraries of a capability, one can use adapt query lib/capability lib and receive a default location (/usr/lib), even if the capability is not installed yet. If this pretended location does not work (i.e., requested libraries are not there), it triggers deployment for this capability and the query repeated will return the new location.

The actual deployment recipes are located in each class and they follow the deploy*.sh naming and ADAPT-D tries them in an order. Optionally, each deployment script can have auxiliary rollback and tune scripts (with postfixes _cleanup and _tune, respectively). The tuning script is the one that handles the various configuration requests. For example, the trilinos library may be built with options that influence how the library is assembled, such as enabling MPI, using boost or HDF5

libraries. To support configurations, the call of `adapt deployment capability` can be extend with options such as `mpi=on hdf5=on`. The users must provide a decoding that can transform these flags into sequences steering the build, e.g., for the `hdf5` option, the users must provide the `opt_hdf5` script that yields `-D TPL_ENABLE_HDF5:BOOL=ON` for a cmake-based recipe and the `decode_options()` method applies it in the recipe. Applied options are saved in `meta` for future references. The error detection–fix information is stored in `ehandlers` files. Examples of a recipe and `ehandlers` are given in listing 6.1 and listing 6.2, respectively.

Listing 6.1: A recipe for LifeV simulations from the `lifev` class

```
#!/bin/bash

# stage-in auxiliary adapt tool
src=$(adapt tool get $GIT_HOME/ABF.tar.gz)
bld=$src/bld
mkdir -p $bld
adapt register self src=$src bld=$bld

cd $bld || exit 1
# if adapt fails, the script exits immediately
adapt cmake ..
adapt make
make install
```

Listing 6.2: The `ehandlers` file for the `lifev` class

```
.*Could not find a package configuration file provided by "LifeV".*
 ~> adapt deploy lib/lifev core navierstokes operator ecm2 hdf5
.*LifeV MUST have ([^.]+).* ~> adapt deploy lib/lifev \L\1
.*error: 'partitionIO' was not declared in this scope.*
 ~> adapt deploy lib/lifev hdf5
```

6.4 Example

In order to test usability of ADAPT-D, we tested it deploying an SaE-class application that is a blood flow (hemodynamic) simulation developed with the assistance of the library LifeV [5]. In this section, we report our experiences with deployment of this software on a single node of a peer-to-peer parallel system—this may be an equivalent of a frond end for a cluster. Our selected target is equipped with Intel i7-2600 CPU (3.40GHz, 4-core with HT), 8GB RAM, has a local file system available for users under /tmp (ADAPT-D uses this directory for as the installation destination and build scratch space), and the home directory is served over NFS. The operating system is CentOS release 6.5 and we did not have a privilege access to the system (unable to use yum). The capabilities offered on the target (the software that is needed during the build) include: (1) GCC v. 4.4.7 (C, C++, and Fortran), (2) the ATLAS library v. 3.8.4-2 (both BLAS and LAPACK) [85], (3) GNU Make 3.81, (4) and typical system tools such as tar, wget, curl.

For our tests we used exclusively the recipe listed in listing 6.1, as currently this is the only available deployment method (this software is not provided in a binary form and source codes are available from the git repository). All dependencies checks are induced by (1) cmake that uses its metainformation about the LifeV and Trilinos libs builds and (2) make. Figure 6.7 presents a fragment of a possible execution diagram for a target described above; all presented actions are commands executed in adapt and the lifelines represent the deployment frames. As it is shown, ADAPT-D induces errors and reacts on them upon analysis of error evidences. The same recipe executes differently if some capabilities are available during this deployment. Also, ADAPT-D changes the deployment paths, if some capabilities are installed but do not provide requested properties. In this situation, ADAPT-D requests appropriate changes adding needed features, e.g., if the LifeV library is not configured with HDF5, ADAPT-D will call adapt deploy lib/lifev hdf5 to add support for HDF5.

For the described configuration, ADAPT-D softconditions that target in about 30 minutes and installs about 500MB of binaries and headers including CMake, OpenMPI, SparseSuite [120], UMFPACK, ParMETIS [117], Boost, HDF5, LifeV, Trilinos, zlib, bzip2, and ANN [121].

6.5 Conclusion

This research describes a prototype of an automatic software deployment system named ADAPT-D. We focus on SaE applications that are considered to be hard to install for the reason that they have usually various and non trivial dependencies, are distributed mostly in the form of source codes, and target heterogeneous computational machines, which often are cutting-edge, experimental architectures. To worsen this unfavourable situation, SaE software is addressed mainly to field scientists who do not have to be (or even do not want to be) well trained computer experts. These issues are mitigated, to some extent, if the users can outsource software deployment to supercomputer center experts. Emerging of powerful workstations, affordable clusters, and cloud computing solutions tempts the users to own these targets to improve availability and increase experimentation with new software. Conversely, using these machines the users must rely only on themselves.

ADAPT aims at providing transparent and adaptive software execution and deployment. In comparison to other softprovisioning techniques that follow a bottom-up approach, ADAPT-D presumes full installation of the application to be executed and reacts on runtime errors when they appear. After the initial recursive error–fix deployment phase, the required software is installed and ready to use—in this manner, ADAPT-D proposes a top-down solution that allows for dynamic detection and adaptive provisioning of missing requirements. We envision that thanks to our project field scientists can experiment with any SaE-class software tools and use them to improve their productivity and innovation. The deployment part of the ADAPT-D project is being actively developed and may be tracked at [146].

6.6 Contribution

Most of application deployment methods are based on providing software layers in the step-by-step manner. For a typical SaE application these layers are: compilers (e.g., C++), basic system libraries (e.g., stdlib), basic scientific libraries (e.g., Blas), more specialized compilers or interpreters (e.g., mpicc or python), specialized libraries (e.g., ParMETIS, HDF5), aggregating libraries (e.g., Trilinos), problem-solving libraries (e.g, LifeV), and the client software (e.g., a specific simulation

code). All components create an acyclic network of software dependencies thus it is natural to proceed provisioning in the bottom–up manner as the upper levels fail without proper support.

To facilitate the deployment phase a few solutions are proposed. One idea is to rebuild a precisely-defined software set from scratch presuming that targets provide no software capabilities (as it is done for VisIt [147]). However, this is time consuming and tends to multiply installed packages. In addition, vendors must handle all possible situation-specific cases and prepare alternative solutions as each resource type may need different set of dependencies. Another approach is to avoid build errors by extensive checks of appropriateness of the build environment using numerous time-consuming probes (e.g., a pedantic check would verify each function from external libraries). The check phase reports inconsistencies in the environment to the users who are responsible for solutions. In this method, software vendors are supported as probes can be generated automatically; however, the users must demonstrate expertise in computer science and devote time to troubleshoot issues. On the opposite side of the spectrum are tools that help nobody and just organize the build steps without providing any support for dependencies (e.g., `make`).

We propose an approach that relieves both vendors and users and takes responsibility for provisioning of the build environment. The software product of our research, `ADAPT-D`, does not avoid build and runtime errors but exploits them as guidelines to enhance the environment of the target. In this sense, we propose an upside-down model where our automatic deployment tool induces errors, analyses them, and reacts on them. On a missing dependency signal, the tool forks to provision the missing software which results in creating a deployment (recursive) provisioning tree. All provisioning steps follow a problem–solution design and this knowledge is generic and reusable as most build errors originate in generic, situation-agnostic tools, such as compilers or linkers. As we follow the top–down approach, this tool is directly applicable for execution as it solves errors in runtime.

Using of `ADAPT-D` dramatically simplifies deployment maintenance as (1) developers of scientific applications can concentrate entirely on their codes and the functionalities that they deliver and (2) end-users do not have to investigate recondite build messages and may concentrate on pure usability of the software. As a result, it should improve productivity in SaE computing and enable experimentation with unfamiliar SaE applications—as they become easy to deploy—and new resource types—as they can be offered unspecialized; this should promote progress in science.

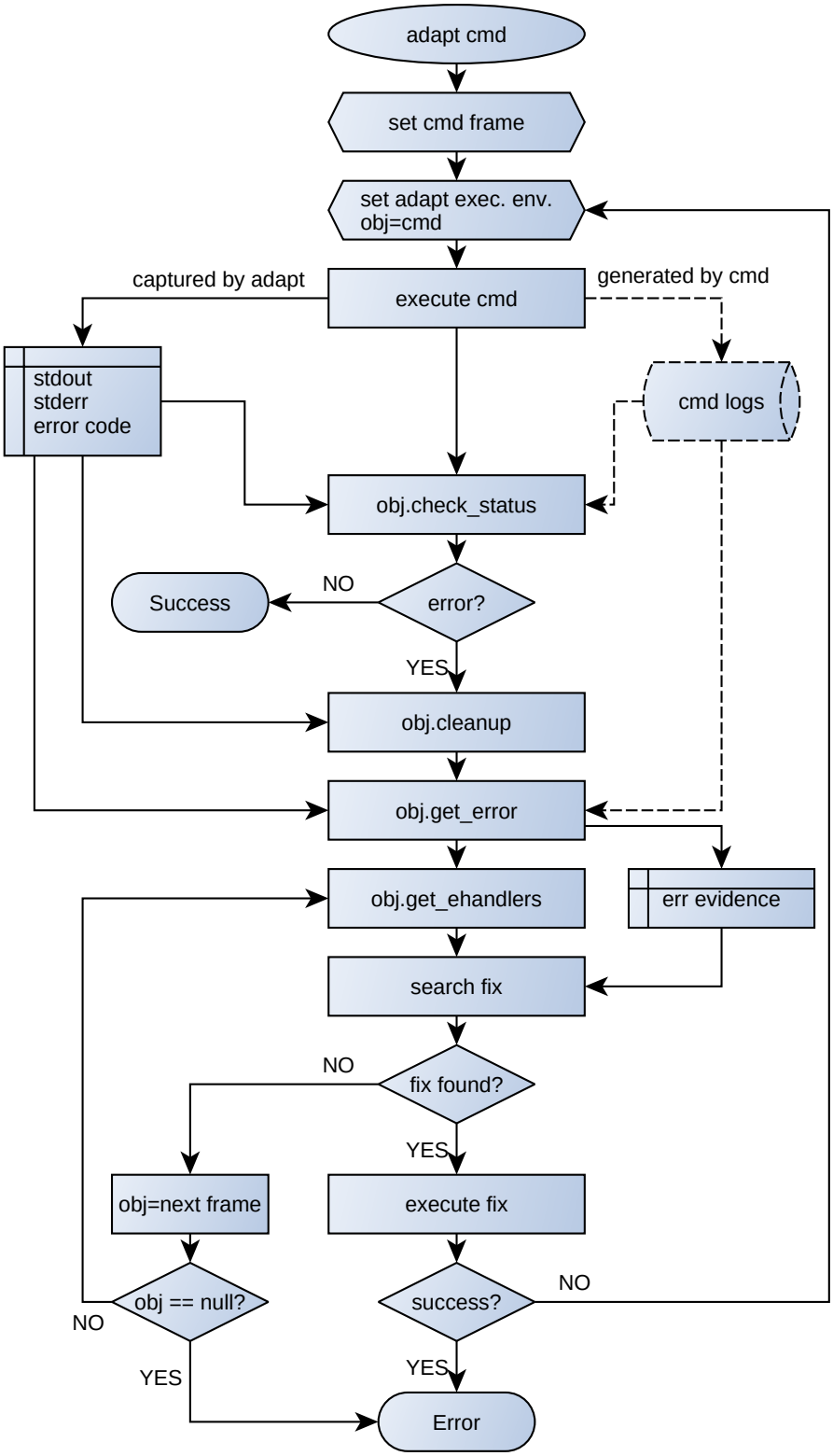


Figure 6.3: Execution of commands in ADAPT-D

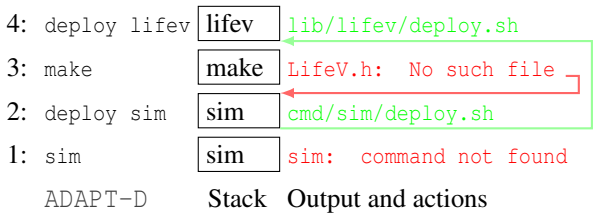


Figure 6.4: Building of the stack of frames. If `sim` is unavailable, `ADAPT-D` deploys it using a deployment script that calls `adapt make`. The `make` frame "knows" how to provide the error evidence for the compilation error; however, the `sim` frame is needed to recognize and react on this specific error.

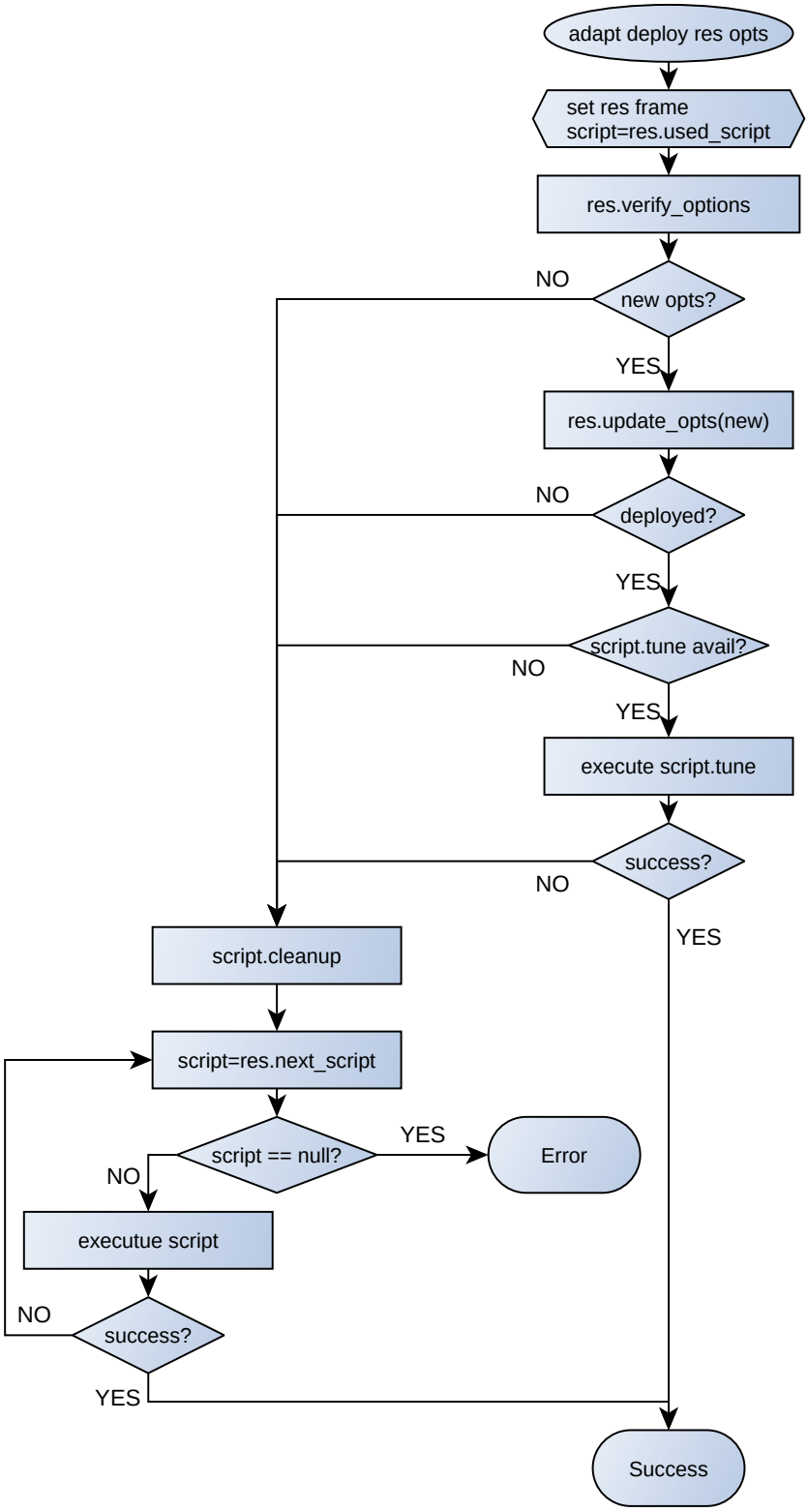


Figure 6.5: Deployment of resources in ADAPT-D

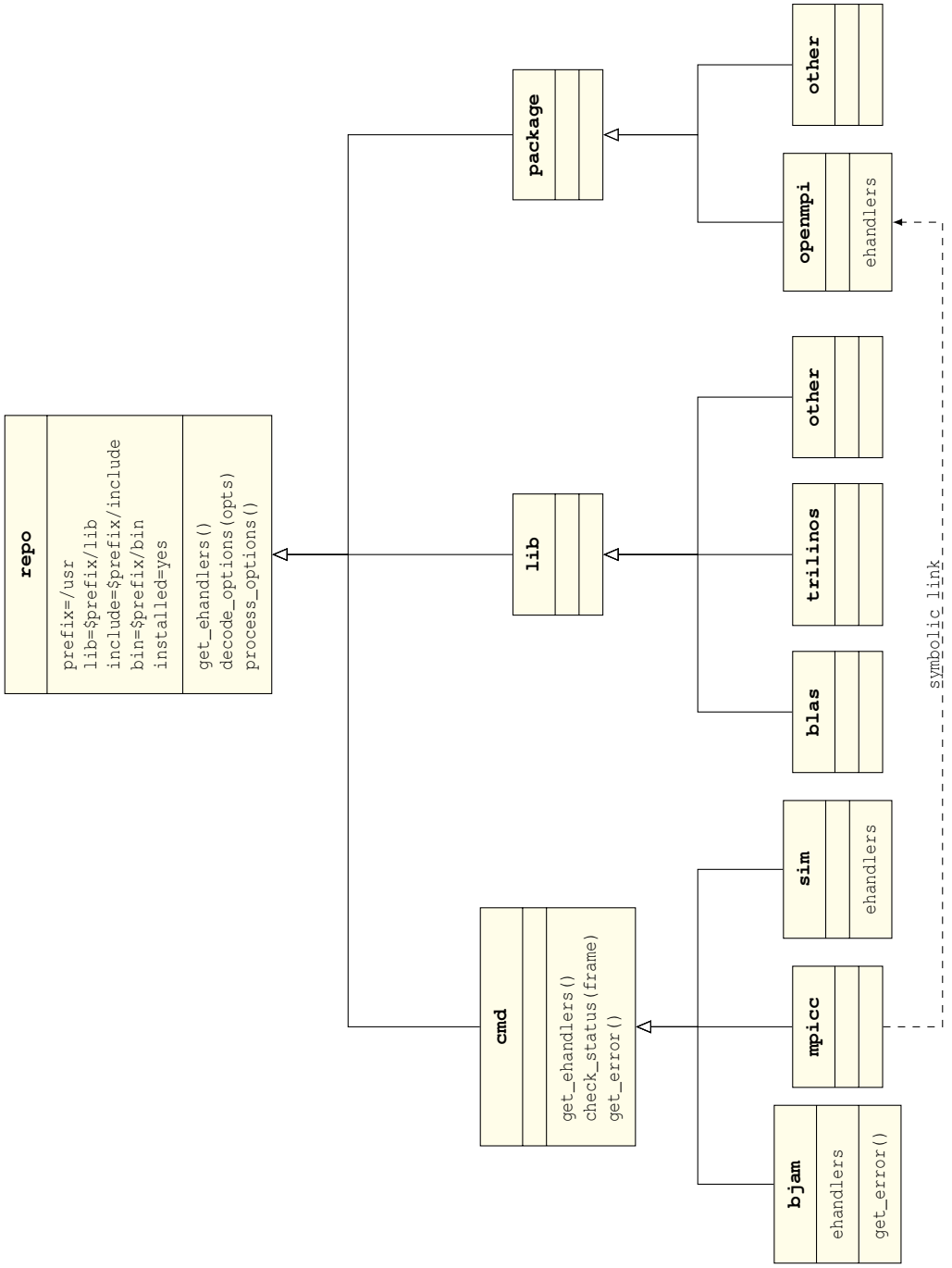


Figure 6.6: An example UML diagram of the ADAPT-D repository

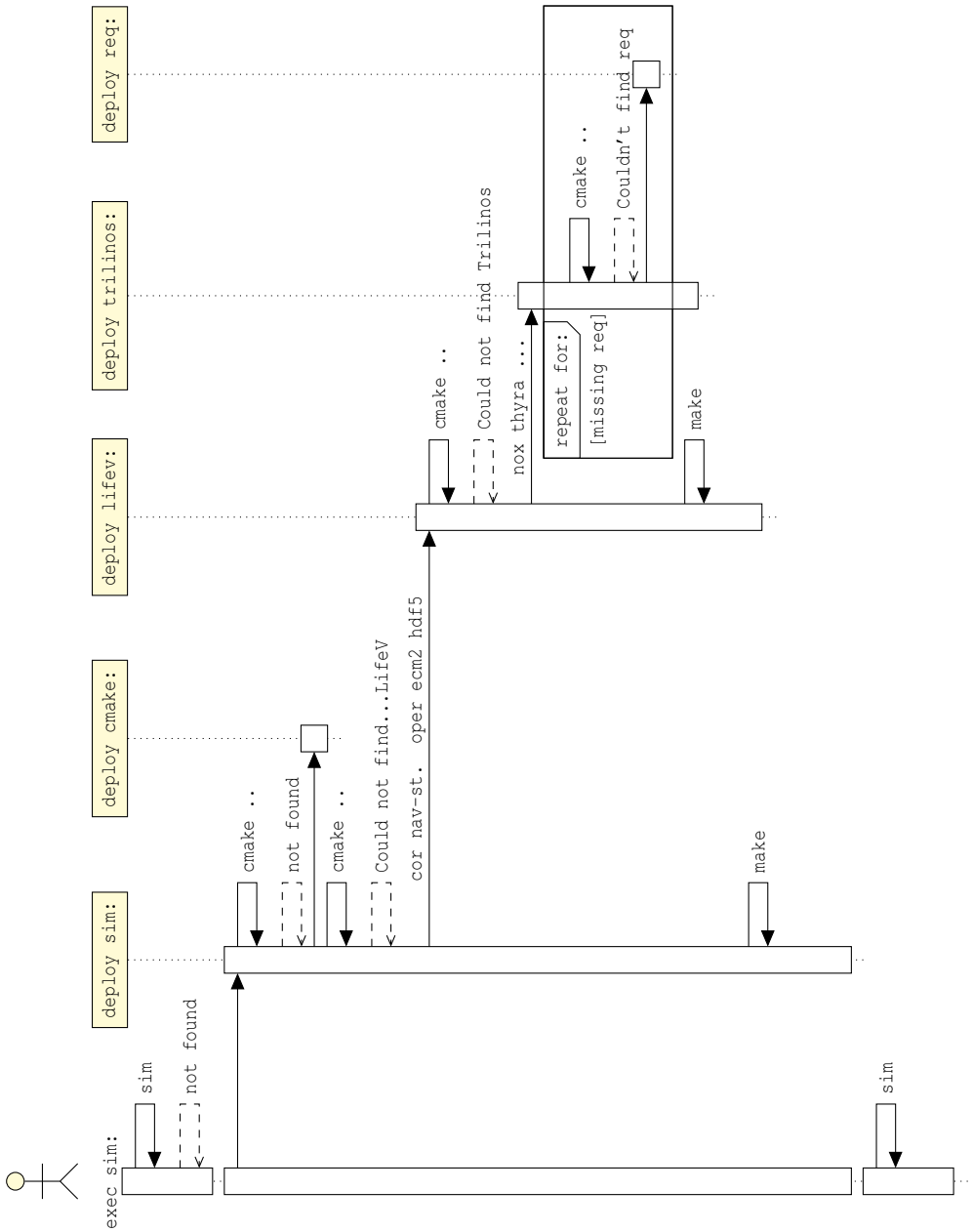


Figure 6.7: A fragment of the execution diagram for the `sim` deployment recipe

Chapter 7

Application Paradigm Adaptation

7.1 Background

The holy grail of computing as a utility has seemingly progressed towards reality in recent years. With the advent of cloud computing, following the era of grid computing, on-demand resource sharing or resource access is already in the mainstream in several domains, suitable for many classes of applications. However, in order to realize true utility-like access to cloud computing, advances are still required along many dimensions. For instance, one desired attribute of utility computing is the ability for any provider's offering to meet any user's requirement but the variety of programming paradigms and platform models make this far from prospect. Especially for parallel or distributed programs, this matching is quite complex.

In cloud computing platforms, typically three widely varying levels of programming model support are provided: PaaS platforms support one, very specific and rigid, programming model, while SaaS and IaaS platforms offer none at all. That is, SaaS platforms are not programmable while IaaS platforms present a typical von Neumann machine, i.e., bare bones, sequential, imperative programming model. The region between IaaS and PaaS programming models is a very interesting one and begs the question of application flexibility in moving between these two end points. If applications written in one IaaS/PaaS/in-between model could be easily adapted to execute elsewhere in the region, an additional degree of freedom can be achieved between providers and users, thereby enhancing the "utility" definition.

Moving from one type of cloud platform to another brings an entirely new set of challenges but it may be argued that abstractions may be layered to address this issue in one direction. For instance, a MapReduce [1] application can be executed on an IaaS platform by first deploying a MapReduce framework, such as the public-domain Hadoop implementation. Services may similarly be launched after conditioning an IaaS platform, although they may not be able to run on PaaS clouds. In general though, higher specializations may be implemented on more generic layers, e.g., SaaS on PaaS, or PaaS on IaaS clouds [34].

We focus on clouds as a utility for pre-existing (legacy) applications and assume such applications are normally run on a (typically on-premises) "native" platform. Executing such legacy *sequential*, *production* applications on a *specific* IaaS platform is straightforward. A one-time effort involves building the application for the target environment under consideration. For example, on Amazon EC2 [98], an image can be selected/tailored that is well-suited to the execution of the application in question, includes the required libraries, etc. In particular, any auxiliary software that is required is identified, selected, and packaged at this time. Deployments of the application on this IaaS platform follow the same procedures as execution on the native platform, subsequent to instantiating the IaaS image instance (or equivalent).

However, moving the application even to another IaaS platform is not at all "utility-like". Often, the application has to be recompiled, with potential adjustments needed to match libraries and environmental artifacts of the new target platform. Occasionally, source code changes may be required. And when applications are under development (as opposed to stable production codes) these inconsistencies are greatly magnified. Finally, when programs are *parallel* or distributed, as many science and engineering applications are, cloud computing is nowhere near as seamless or transparent as a utility must be.

We attempt the inverse—deploying procedural message passing programs on a MapReduce platform—as one of the adaptations we intend to explore in the `ADAPT` project. Although begun as an academic exercise, our experiences provide several insights into the feasibility of such a mapping and highlight some collateral benefits of deploying certain classes of MPI applications on MapReduce platforms. More generally, this potential for cross-paradigm execution marks a characteristic in the utility-like nature of cloud computing. Our approach is based on the concept of adapters, common in traditional utilities, to reconcile application requirements to platform facilities.

We consider the scenario of executing simple MPI programs on a PaaS cloud that presents a MapReduce interface. We show the transformations and shims that can enable such porting. As a first approximation, much of this work was done manually; although understanding the issues may be helpful in automating such processes in the future. We also acknowledge that the scenario may seem contrived; the rationale for executing an MPI application on a PaaS platform is likely purely academic. Nonetheless, the experiences gained through this exercise provide interesting insights into the relationships between common programming models used in current and emerging cloud platforms.

7.2 Related Work

Cross-platform portability has long been a focus of intensive research, and is a key factor in making grids and clouds utility-like. One approach is resource homogenization, where a standardized or uniform environment is created, or emulated, on target platforms. This model has been extensively explored since leads quickly and in a straightforward manner to portable applications. Standards and commonly used toolkits, such as Globus [29], greatly facilitate the use of the underlying computational resources and allow for seamless application execution. Standardization efforts in progress, such as the Open Cloud Computing Interface (OCCI) from the Open Grid Forum (OGF) [148] have the potential to help migration of applications. Example implementations include OpenNebula [149] and Eucalyptus [150]. Even simple resource-access standardization, as proposed in projects like Simple Cloud API [104], Nimbus [31], or AppScale [32], make executing applications on different targets easier. However, there are two major drawbacks to this approach. First, they often require re-programming of the application and/or modifications to express the application suite in terms of the standard API. Legacy applications are therefore not supported "out of the box." Second, standards layers conceal the unique, but possibly valuable, traits of individual target platforms and underlying resources. `ADAPT` aims to allow access to (or even expose) the full potential of the computational resource while conditioning the target's capabilities to those required by the application, extending these capabilities where needed. The adapter model also facilitates executing of applications without modification.

At a different level, software frameworks and execution environments, like Hadoop [151, 86], MPICH [87], or Matlab [88], present programming model standards, thus providing homogenization at the application framework level. This reduces or eliminates resource dependencies but implies some effort for resource vendors or operators in provisioning the given environment. The issue of legacy applications and those developed in a different programming model remains. The goal of `ADAPT` is to alleviate both resource and programming model constraints by transforming application dependencies to the facilities provided by computational back-ends through the use of adapters. A similar approach is adopted by projects such as PortableApps [107] and rPath+rBuild [37].

In this experiment, we explore the possibility of executing MPI applications on a MapReduce infrastructure. The inverse problem—running a MapReduce application in an MPI-native environment—is also a research topic that has received attention; projects that have explored this include MapReduce-MPI [18, 89]. The essence of our approach is to divide the execution of an MPI application into a sequence of computation jobs, each terminated by a communication phase. As such, each job functionally resembles a *superstep* in the Bulk Synchronous Parallel (BSP) [90] model. In fact, BSP-MapReduce libraries, such as Apache Hama [152], are comparable to our work at a certain level. Similarly, iterative MapReduce frameworks, e.g., Apache Mahout [91], Twister [92], or Pregel [93], follow the same BSP idea—they provide a software scaffold to easy implementation of repetitive algorithms, such k-means clustering or graph processing, and preserve the properties of the underlaying MapReduce paradigm. Comparing those with our approach, we target the generic, algorithm-agnostic MPI application execution by pressing the MPI paradigm into the MapReduce framework. However, we can substitute the actual generic, low level MapReduce implementation and build upon those solutions. This substitution may simplify our software by eliminating routines responsible for chaining separate MapReduce jobs. In addition, the features of underlaying frameworks (e.g., improved performance over the bare MapReduce) may enhance MPI application execution.

Another related research area concerns frameworks that merge features from two or more platforms; an example is CloudBLAST [94] that combines MPI and MapReduce to provide a parallel SaaS environment for bioinformatics.

7.3 MPI-MRE Execution Environment

In this work, we present an exploratory *paradigm shifting* use case of adaptation, viz. execution of simple *unmodified* MPI applications on the Hadoop MapReduce infrastructure (MR). As our project is in an early phase, this exercise is restricted to MPI programs that only use collective communication calls, i.e., those that do not use point-to-point send and receive calls. We describe the ADAPT scaffolding termed MPI-MRE that enables execution of MPI programs in MR Environments and report on logistical, performance, and overhead observations.

7.3.1 Idea

The key challenge to overcome in our proposed solution (MPI-MRE: MPI on Map–Reduce Environments) concerns the essential characteristic of any MPI application, viz. addressable data exchange between independent, interacting processes—a feature unavailable on typical MapReduce infrastructures. Thus, in order to execute MPI applications in MR environments, the missing capability of addressable data exchange must be assembled, or composed, from existing capabilities; such a concept is shown in Figure 2.6.

Comparing the key characteristics of the MR and MPI paradigms suggests the following translation: (1) each MPI process should be mapped onto a single MR mapper and (2) data exchange must be accomplished between the map and reduce phases. This further implies that (3) communication operations should be split into three phases, with the "sending" phase in one set of MR jobs (mappers), the "communication" phase embedded in the reducer portion of this set of MR jobs, and the "receiving" phase in the re-incarnated set of MR mapper jobs, as depicted in Figure 7.1. That is, the first set of mappers (MPI processes) initialize the communication, then the MR mechanisms shuffle and stream the data to the reducer(s) that re-instantiate another set of mappers that receive the exchanged data and continue the MPI processes. Such a scheme points to the conclusion that a (4) MapReduce back-ended execution of an MPI application can be organized as a chain of MR jobs. The limitation of such a design is that currently only collective communication operations are supported due to the synchronization requirement to advance the chain of MR jobs—all mappers and reducers must complete to finish a job and start the dependent job.

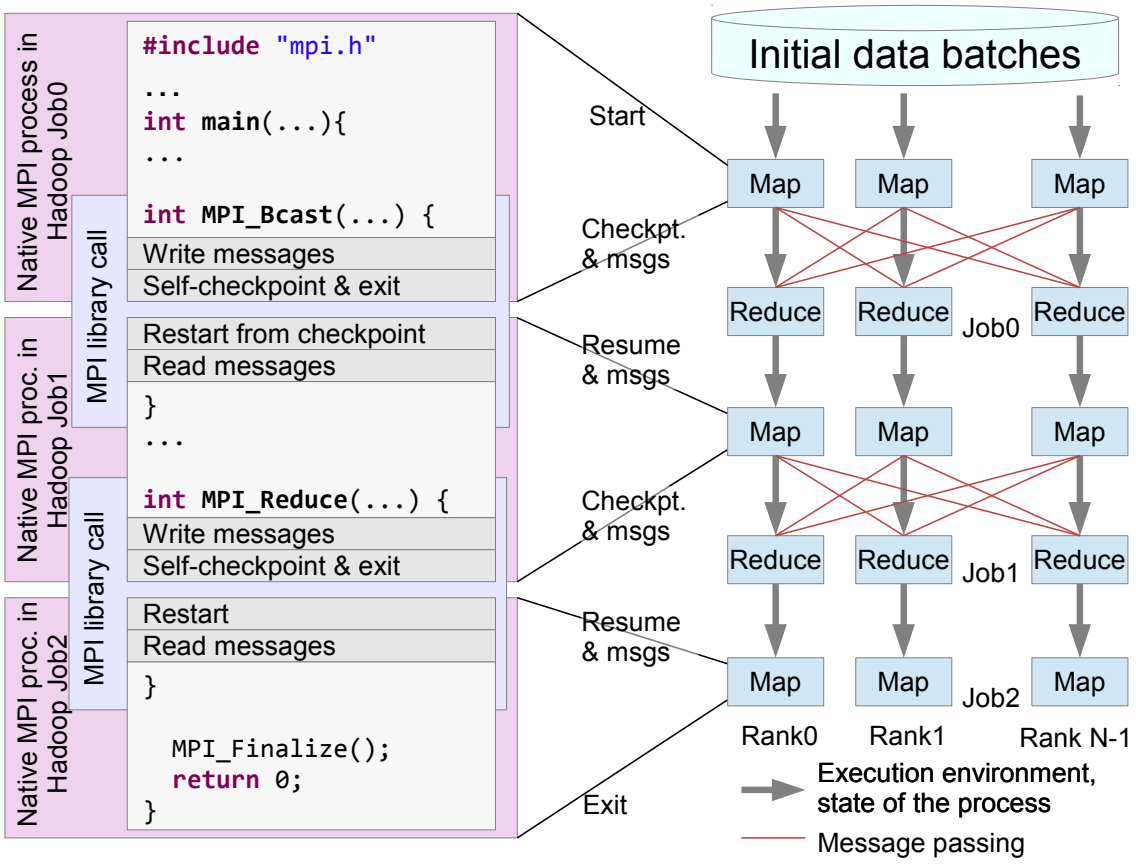


Figure 7.1: The `MPI-MRE` runtime; execution of `cpi`

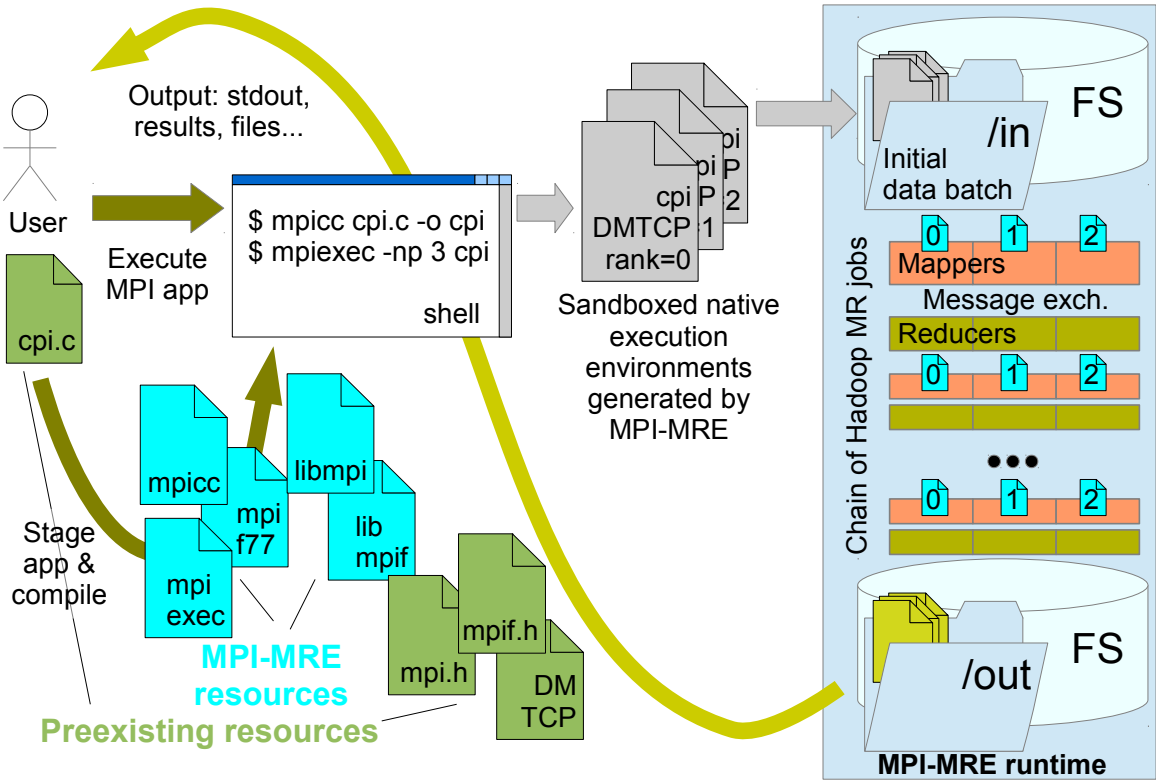


Figure 7.2: The MPI-MRE operational scenario

Our implementation solution, depicted in Figure 7.2, consists of (1) MapReduce-based MPI library implementation, (2) Hadoop-based Java runtime middleware, and (3) MPI execution environment emulation scripts. This adaptation is an example of cross-paradigm execution that is achieved through a combined adjustment of both the application requirements and target capabilities. In this particular implementation, we form the communication capability required by MPI applications but not necessarily exposed by the Hadoop-based resource; in essence, we exploit the Hadoop Distributed File System (HDFS) storage capability to deliver messages to the MPI processes. The following paragraphs describe the design and implementation of these constituent components.

7.3.2 MapReduce MPI Library

A mandatory dependency for any MPI application is the MPI library. The ADAPT project provides an MR implementation of the MPI library that mediates between the MPI application and the

underlying MPI-MRE runtime layer, which manages execution, provides the communication capabilities (e.g., using storage for data exchange), and carries out the actual data exchange. To enable message exchange in MPI-MRE, the library and the MPI-MRE runtime component use local temporary storage: the library passes the signature and arguments of the communication routine being processed by serializing them into an MPI rank-related file; in a complementary manner, the MPI-MRE runtime passes the exchanged data back to the recipient in accordance with the routing specification. Moreover, any operations on the received data (i.e., specific operations to apply during reduction, such as a global sum) are conducted in the library to avoid possible marshaling issues. Such a separation enables the MPI-MRE library to be agnostic of the MapReduce framework. Execution therefore only requires that the MPI application is linked with the MPI-MRE library. As the result, our solution is applicable both to source codes, which need to be compiled against the library, as well as to dynamically linked binaries.

Recall that MPI-MRE communications are split into back-to-back MR jobs as described above. An MPI process embedded in the anterior MR job is reinstated in the posterior job, straddling the data exchange. This suspension and re-instantiation is implemented using a checkpointing library (currently we use DMTCP [95]). At the conclusion of the MPI process, the library does not generate a checkpoint file—this signals the runtime to cease the chain of MR jobs and exit. Note that the checkpoint library constitutes a dependency for adaptation.

7.3.3 `MPI-MRE` Runtime

The runtime subsystem is the core of our solution as it assists in MPI process execution and implements the actual data exchange. Since we decided to utilize the Hadoop framework for MapReduce in this exercise, we followed the design of the framework and developed the MPI-MRE runtime in Java to have native control of Hadoop jobs.

Hadoop organizes computations as MapReduce jobs. Mappers are responsible for launching (or restarting from the checkpoint) the native MPI processes, acquiring and delivering MPI messages as well as forwarding the paused process execution environments to the next processing phase. The Hadoop shuffle stage performs the actual delivery of messages: all data labeled by the unique MPI rank ID are exchanged and transported to reducers.

The reducers merely need to provide a mechanism for collating the digested data and write the data to be used as input for the following mapper phase. We overlaid the MPI-MRE runtime on this structure as follows. Each MPI-MRE Hadoop mapper first receives a package of data that contains the *sandboxed execution environment* for a *native MPI process*. We create as many initial standalone data packages as the number of MPI processes requested. After streaming the data in, the mappers prepare the data for the initial MPI calls (`MPI_Init`, `MPI_Comm_size`, and `MPI_Comm_rank`) and spawn the native MPI processes. When this process completes and releases control after the self-checkpoint, the mapper receives the arguments from the communication call and passes the message(s) to the reducers in accordance with the MPI routine specification. To avoid marshaling issues and promote logical separation, the MPI-MRE runtime supports only message exchange, leaving the actual data processing for native MPI processes. Along with the message data, the mapper passes the sandboxed environment files, any initial data or generated files, the checkpoint file, and serialized standard streams.

Each reducer receives (1) the complete set of native execution environment files for its MPI rank and (2) destined for it messages from other MPI processes. Next, the reducer combines these messages into a single file that will be conveyed to the resumed native process. The last step of the reducer is to prepare a single file including the complete native execution environment. This file is saved in the MR job output directory and becomes the input for the next MR stage. This cycle stops when the MPI-MRE runtime detects that the output does not contain a checkpoint file (indicating that the native MPI process exited normally, not in order to process a consecutive MPI communication operation).

7.3.4 The MPI Environment

The experimental implementation of MPI-MRE also contains a few simple scripts to mimic traditional MPI environments. `mpicc` and `mpif77` support the compilation of MPI source codes against our MPI and DMTCP checkpoint libraries, satisfying the dependencies necessitated by our framework. Another script, `mpiexec`, accepts user parameters, prepares and stages-in the *initial data batch* into the Hadoop input directory, and launches the MPI-MRE runtime. In addition, `mpiexec` handles the standard system output streams. The initial data batch is the set of identical sandboxed execution environments identified by unique integers that become MPI process ranks.

NP	cpi	vv_mult	EP.A	EP.B
1	102.9	303.0	254.1	304.0
2	104.9	303.4	246.3	270.8
4	114.7	331.0	262.8	276.8
8	129.1	368.4	306.7	327.4

Table 7.1: Execution times in seconds for test applications on MPI-MRE

Each environment package includes all files required to start the native MPI process and to commit the checkpoint. When the native MPI process suspends, the sandboxed package is updated. The number of execution environment packages is conserved during the entire processing. This operational scenario is depicted in Figure 7.2.

7.4 Results

We conducted one set of tests for the basic cpi application on the Amazon Elastic MapReduce (Amazon EMR) [100] service, an exemplar of PaaS clouds. To mimic a native MPI execution environment on Amazon EMR, MPI-MRE permits users to interact directly with the Hadoop namenode host, but our solution also works with the native EMR service interface that exposes its capabilities through the Amazon Simple Storage Service (S3).

However, the design of our framework enables its use on any resource powered by the Hadoop (or equivalent MR) framework. Therefore, for convenience of experimentation and to avoid factual costs, we conducted the tests on a single machine equipped with one 4-core Xeon CPU (E3-1225 at 3.10GHz), 20GB RAM, and single SATA HDD (7200 RPM) running CentOS 6.2. We used Apache Hadoop 1.0.2 run in Pseudo-Distributed Mode. Configuration parameters (applied in conf/mapred-site.xml) set the Java memory heap to 2GB, allowed parallel execution of up to 8 mappers/reducers, and rescinded the default Hadoop task timeout. Finally, we used the tar command to bundle the execution environment files into a single input file for the MPI-MRE runtime.

We tested three SaE-like textbook applications: (1) the C program cpi included in the MPICH distribution [87, 153] that computes the value of π by the numerical integration method (i.e., it solves $\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$ with the step $\frac{1}{10000}$), (2) the C-based vector-vector multiplication code (vv_mult, available on-line as a file vv_mult_blkstp_unf.c) that reads two vectors from the input files, and (3) the standard Fortran EP test from NAS Parallel Benchmarks [15] that generates random values

using the Marsaglia polar method. For the latter code, we tried four size classes: A, B, C, and D (they generate 2^{28}, 2^{30}, 2^{32}, and 2^{36} random variables, respectively).

	number of processes			
environment	1	2	4	8
		EP.C		
OpenMPI	280.7	141.4	71.7	74.1
MPI-MRE	517.2	378.6	327.4	375.9
		EP.D		
OpenMPI	4488.8	2245.1	1124.1	1133.6
MPI-MRE	4733.1	2485.4	1385.4	1435.7

Table 7.2: Comparison of execution times in seconds for EP.A and EP.B on MPI-MRE and OpenMPI

Listing 7.1: Skeletal code of cpi

```
#include <mpi.h>
...
int main(){
 ...
 MPI_Init();
 MPI_Comm_rank(&myid);
 ...
 n = 10000;
 ...
 MPI_Bcast(&n, 1, MPI_INT, 0);
 // do the kernel
 MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0);
 if (myid == 0)
   // root does the output
 ...
 MPI_Finalize();
}
```

The skeletons of the codes showing only the MPI communication-relevant sections are included in Listings 7.1, 7.2, and 7.3. The simplest code, cpi, uses only two collective operations. The vector–vector multiplication application uses eight MPI collective functions (note: the first four MPI operations in the original code are somewhat superfluous but we retained them in the interest

of not modifying applications). Finally, the EP code employs five collective operations. Note that though MPI_Barrier does not exchange any messages, it has to be fully processed by the MPI-MRE and must be treated as a collective call. We executed cpi, vv_mult, and EP with the assistance of our framework to determine and demonstrate the viability of this approach. All three applications executed without incident and produced valid and correct results. A natural question in any endeavor, such as ours, concerns performance overhead. Indeed, it is self-evident that forcing synchronization, process checkpoint and restart, and routing all message-passing via disk storage will incur considerable expense. However, our project is aimed at complete flexibility even though it may come with some performance penalty and, therefore, we argue for its usefulness at least in certain situations. Nonetheless, it is useful to understand the magnitude of the overheads involved. During the experiments described above, we measured elapsed times using the time mpiexec -np NP app command where NP was 1, 2, 4, and 8. Speedup is bounded by the number of cores in our test platform (4) but tests with 8 processes were included to observe how the MPI execution environment behaves for excessive load. We repeated all tests twice and averaged the results.

Listing 7.2: Skeletal code of vv_mult

```
#include <mpi.h>
...
main(){
 ...
 MPI_Init();
 MPI_Comm_rank(&MyRank);
 MPI_Comm_size(&Numprocs);
 if(MyRank == 0)
   // read vectors vA and vB
 MPI_Barrier();
 // spread info about read vectors
 MPI_Bcast(&vA_OK, 1, MPI_INT, 0);
 MPI_Bcast(&vB_OK, 1, MPI_INT, 0);
 MPI_Bcast(&vvOK, 1, MPI_INT, 0);
 MPI_Bcast(&vLen, 1, MPI_INT, 0);
 size = vLen/Numprocs;
```

```
 MPI_Scatter(vA, size, MPI_FLOAT, myA, size, MPI_FLOAT, 0);
 MPI_Scatter(vB, size, MPI_FLOAT, myB, size, MPI_FLOAT, 0);
 // do the kernel
 MPI_Reduce(&MyFinVec, &FinAns, 1, MPI_FLOAT, MPI_SUM, 0);
 if (MyRank == 0)
   // root does the output
 MPI_Finalize();
}
```

Table 7.1 shows the execution times of `cpi`, `vv_mult`, `EP.A`, and `EP.B` run in our `MPI-MRE` framework. Since these applications are not computationally intensive for contemporary CPUs, the times mainly represent the overhead of communication routines implemented in `MPI-MRE`. Execution times for each application are very close irrespective of the number of processes used. This further suggests that runtimes are completely dominated by overheads and any speedup due to parallelization is shrouded. When moving from 1 to 2 cores, in the instance of `EP.A` and `EP.B`, some small speedup is noticeable, which is consistent with greater computational demands of `EP`.

Listing 7.3: Skeletal code of `EP`

```
  program EMBAR
  include 'mpif.h'
  ...
  call mpi_init()
  call mpi_comm_rank(node)
  ...
  dp_type = MPI_DOUBLE_PRECISION
  ...
c synchronize all processes
  call mpi_barrier()
c do the kernel and verify
  call mpi_allreduce(sx, x, 1, dp_type, MPI_SUM)
  call mpi_allreduce(sy, x, 1, dp_type, MPI_SUM)
  call mpi_allreduce(q, x, nq, dp_type, MPI_SUM)
c get the max exec time
```

	number of processes			
	1	2	4	8
time [in sec.]	46.6	46.5	50.7	58.8

Table 7.3: The average overhead of the MPI collective operations

```
  call mpi_allreduce(tm, x, 1, dp_type, MPI_MAX)
  if (node.eq.root) then
c root does the output
  endif
  ...
  call mpi_finalize(ierr)
  end
```

In order to try and better characterize the overheads of MPI-MRE, we ran bigger versions of EP—class C and D—and compared them with their corresponding OpenMPI executions. Table 7.2 shows the execution times achieved on both MPI-MRE and OpenMPI. As expected, the much larger computational load hides the excessive MPI-MRE overhead; indeed, for class D, the timings are comparable.

Finally, using the communication-aware application skeletons from the Listings (number of MPI operations) and the execution times, we estimate the overhead of a single MPI-MRE collective communication operation. Although this time is a function of the volume of data to be exchanged, we ignore this factor for this estimation, given the relatively small messages involved. Also, the startup and teardown of the MPI-MRE mpiexec add to the overhead—preparing the sandboxed execution environment and managing the output data increase the overall time. Table 7.3 shows the averaged overhead time per MPI collective operation in MPI-MRE, including the amortized initiation and termination costs.

Due to the nature of MPI-MRE, the estimated overhead times are mainly due to the Hadoop MapReduce framework: staging the data to/from HDFS, starting separate jobs, and shuffling data from mappers to reducers. Additional, but relatively little, time is required by the checkpoint mechanism. However, the checkpoint files have to be transferred to HDFS in order to perform the restart which is a time-consuming process. Note, however, that thanks to overlapped Hadoop mappers the communication overhead increases only slightly with number of processes.

7.5 Discussion

The proposed solution to the execution of unmodified MPI applications on MapReduce infrastructure may seem artificial or even outlandish, since it is an attempt to project a generic application paradigm onto a narrower paradigm. However, such a composition has the effect of exposing certain desirable properties inherited from the underlying execution back-end. MPI applications that are run on MPI-MRE become automatically *failure resilient* since Hadoop restarts failed components by default. Another feature of Hadoop that can possibly be beneficial is *speculative execution*. Hadoop can be configured to execute replicated copies of tasks when resources are abundant. Depending on the circumstances, earlier completion is possible, minimizing the overall time required. Hadoop also implicitly provides a balanced task mapping mechanism, allocating work to available resources as they become available and queuing work as needed. Both features have the effect of built-in *load balancing*.

Considering the performance aspects of the solution, it is obvious that MR layer will introduce significant overhead. This is especially acute as the time for execution in MPI-MRE is related to the number of communication invocations. As a result, MPI-MRE may be considered a viable option only for a limited class of applications, e.g., parameter sweeps and embarrassingly parallel codes. We believe, however, that a different MR-compatible implementation, e.g., one that is not based on a shared filesystem [96], would significantly facilitate the execution.

Operationally, our solution organizes the MPI application execution as a sequence of MR jobs that handle the parallel processes as the embedded native execution threads. Each process in the job performs its local computations independently. When a communication primitive is encountered, the process *emits* the data to be sent and suspends the primitive—terminating the mapper with a checkpoint of the native process. The reducer phase re-organizes the emitted data that then become inputs to the next set of mapper jobs—which complete the receive portion of the communication. Given the close similarity of this scheme to the BSP model, we plan to evaluate BSP realizations as platforms to support MPI computation. Another intriguing research issue concerns the feasibility and scope for mapping the tightly coupled computation model of MPI onto the unrestricted BSP model. In the extreme, such an investigation might lead to effective MPI execution on poorly coupled cloud architectures [97].

From a broader perspective, ADAPT adopts the view that efficiency issues can be at least partially addressed by engineering enhancements and various optimizations; our current focus is on the feasibility of seamlessly adapting applications to target resources. We believe that this ability is an essential component of the eventual manifestation of *Computing as a Utility* where users are able to launch their applications *unmodified* on any resource chosen from the continuously increasing set of offerings. The current experiment suggests that even unconventional application-target pairs are possible and may even become a valuable choice, e.g., to exploit certain desired features or for other logistical, practical, or expedient reasons.

7.6 Contribution

We proposed and tested a novel approach to enhance executability of SaE applications on heterogeneous resources that eliminates porting and enables using of modern computational platforms even if they do not support application execution paradigm. The foundational idea is to rebuild, emulate, or outsource missing capabilities in a layer-by-layer, fully automatic, and application-specific manner. Our solution applies software transformations, or adapters, that "re-specialize" selected targets; these adapters can serve as virtualized resources or services (e.g., sshd), software shims that mediate between different API's (e.g., different numerical precision), functions' aggregators (e.g., collecting distributed services functions in a DLL), binary translators (e.g., handling of binary incompatibilities), runtime emulators (e.g., providing a command line interface), etc. We also allow for programming language (syntactical) translation adapters, such as a Python to C translator that may enable execution of unmodified C-based applications on execution platforms that support only Python codes. If a capability cannot be recreated from present capabilities, we propose supplying it remotely (outsourcing). After applying adapters, a selected resource is ready to sustain execution of an unmodified application and interact with users in a typical way for this application type (e.g., emulation of an interactive shell for resource accessible via a web service).

The adapter-based approach is broadly accepted for traditional utilities (e.g., power supplies, travel adapters, outlet socket adapters, acoustic modems, optical isolators). In our approach, we never alter applications and never request particular specialization of resources (as it is in traditional

utilities). A situation-specific set of adapters mediates between software requirements and hardware capabilities which enables the execution of the application. A drawback of the method (also similar to traditional utilities) is that a part of capability must be utilized to sustain this adaptation, e.g., data transformations, augmented routine calls, or extra latency for outsourced capabilities; however, novel platforms provide also new, attractive features, such as fault tolerant execution, multilevel caching, or load balancing. As adaptation is built on these features, they automatically enrich the execution paradigm of legacy applications, e.g., an unmodified MPI parallel application may become fault tolerant if executed on MR. We envision that the users might select any resource that meets the minimalistic basic requirements and a toolkit performs automatic adaptation. Thanks to that, the users may switch between or aggregate computational "power" providers based on own preferences which is the fundamental element for the utility-computing vision.

As the result of our considerations, we deliver proof-of-concept software `MPI-MRE` that includes MPI C and Fortran libraries for Hadoop and MPI command tool wrappers to emulate a typical MPI-based developing and executing environment.

Chapter 8

Summary

In this dissertation, we cope with the challenging problem of improving application execution on a broad spectrum of heterogeneous computational resources, with particular focus on Scientific and Engineering software. Streamlined switching between computational resources in order to select the most suitable computational environment for application execution is a desirable, but yet unattained characteristic of utility-like computing. Cloud computing, as computing grids previously, promises realization of this vision but, in reality, new computational offerings have increased heterogeneity—an antithesis to the utility model. So far, already proposed paradigms that span or homogenize various resource types require application porting, which is infeasible for scientific software. Scientific applications have usually specific hardware and software requirements, are often brobdingnagian legacy codes developed for decades, and are tuned for particular architectures to deliver the best performance which severely limits any modifications. Conversely, a number of modern computational offerings experiment with execution paradigms and, as a result, they disallow direct execution of applications using classical paradigms and assumptions. This additionally limits usability of modern resources even if they meet elementary requirements of applications, i.e., have enough RAM or are binary compatible.

Currently, coupling SaE applications to heterogeneous computational targets requires expertise and enormous manual effort. To answer this, we propose `ADAPT`—a novel approach that simplifies deployment and execution for diverse machines. Our idea concentrates on software adapters that are used to create situation-specific, i.e., tailored for the application–target pair, middleware. As this composition can be generated automatically, based on the target capability and application

requirement specifications, this method enables execution of unmodified applications on raw computational resources. In practice, we achieve the coupling through software transforms delivered by adapters and applied on runtime environments.

To verify our propositions, we selected three orthogonal dimensions that hinder usability of different types of computational targets and tested applicability of ADAPT. First, we experimented with performance of parallel applications and modeled it based on utilization of communication links. We showed that it is possible to improve performance by adapting execution of a parallel application for a given resource. We achieved improvement in time-to-completion by logical communication pattern to physical interconnect topology matching that (1) reduces use of long-distance connections causing bottle-neck communication problems and (2) maximizes core-to-core in the same affinity zone data exchanges. In the experiment, we studied a hemodynamic simulation application that analyzed flows in blood vessels. In this case, highly-unstructurized input makes computation difficult and unorganized and we showed that for this particular performance adaptation the matching must consider not only application and physical topologies but also imbalance introduced by the input problem. However, we confirmed that performance tuning for the particular input does not require benchmarks to collect communication statistics. It was demonstrated that it is possible to estimate communication differences based on the geometry of simulated objects. As the grand result, our methodology enables resources atypical for science for SaE application execution as after automatic performance adaptation they can match characteristics of HPC machines. This, in turn, provides more execution options for end-users.

In another set of experiments, we further explored this claim and studied tradeoffs between performance and cost for a few alternative situations and demonstrated that for some operational scenarios using IaaS cloud platforms may surpass benefits of using supercomputers for scientific and engineering applications.

Another problem strongly related to usability of different types of platforms is SaE software deployment. Scientific applications are extremely hard to deploy and often these troubles forbid switching between targets even if logically they are good candidates for execution. To facilitate the deployment phase, we devised original methodology and developed the ADAPT-D toolkit that automatically provisions computational resources in an autonomic manner. In ADAPT-D we fully used ADAPT ideas and introduced deployment recipes, i.e., software adapters, that are adaptably

selected for specific application–target situations to compose middleware. This composition soft-conditions computational resources in a layer-by-layer fashion until a proper capability–requirement matching is obtained. Moreover, adapters capture expert knowledge related to solving particular software deployment issues in one situation and can be reused later in another deployment context which greatly simplifies the maintenance of this phase. We emphasize, that our methodology does not introduce another deployment mechanism but is generic and abstract for existing tools and installation mechanisms. In the context of SaE software, this is of critical importance because scientific applications do not fit well into common deployment schemes that are developed for user applications and diversity in SaE deployment methods is significant. In the context of utility computing, using of `ADAPT-D` supports switching between targets and encourages scientists to experiment with new software and unfamiliar types of computational targets. As a result, we hope that our research and tools have potential to increase progress in science and engineering.

In another constructive investigation, we demonstrated application–platform paradigm coupling. In particular, we used `ADAPT` methodology to enable execution of unmodified parallel applications on the Map–Reduce infrastructure. As a proof-of-concept, we have developed MPI library and MPI tools, named `MPI-MRE`, and tested them on the Amazon Elastic MapReduce Platform as a Service infrastructure that provides computational resources for Java-based Hadoop programs. In a serious of adaptations that transform resource capabilities, we recreated a communication substrate that is unavailable on Hadoop but is necessary for MPI. To permit execution of unmodified applications developed in C and Fortran, we treated their executable binaries as input files copied in the Hadoop Distributed File System storage. Despite `MPI-MRE` is MPI application-agnostic, currently we limited usability of our framework to MPI applications that employ only collective operations; however, it does not diminish the conceptual value of our explorations. One of practical outcomes that were first predicted in `ADAPT` research and now are confirmed is that adaptation preserves valuable resources attributes; this is in opposition to other concepts that, in order to enhance executability, homogenize resource capabilities. In particular, we observed that execution of MPI applications became fault tolerant as native Map–Reduce features transparently enhanced the runtime. In a broader sense, `ADAPT`-based paradigm matching benefits users as it broadens a set of resources that can be used for execution. This increased freedom may lead to new execution paradigms that mashup several paradigms to provide more robust execution model.

We believe that our research ideas, examined adaptations, and delivered proof-of-concept software is impactful and may improve usability of different types of computational platforms for scientific application execution. As the main goal is to avoid application porting, which is the major source of difficulties for SaE software, we focus on software transformations for the runtime environments of computational targets. This augments a set of resources that can be considered by end-users as `ADAPT` methods enable smooth execution of their applications there. It results in increased flexibility as it requires minimum effort from users to switch between suitable platforms for logistical, performance, or financial reasons. Finally, as we attribute this flexibility to utility computing, the `ADAPT` project may help attain this attractive, but yet unattained, idea.

Bibliography: Books and Journals

[1] Jeffray Dean and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113. ISSN: 0001-0782.

[2] Blair Bethwaite, David Abramson, Fabian Bohnert, Slavisa Garic, Colin Enticott, and Tom Peachey. "Mixing Grids and Clouds: High-Throughput Science Using the Nimrod Tool Family". In: *Cloud Computing* (2010), pp. 219–237.

[3] Robert W. Numrich and John Reid. "Co-Array Fortran for Parallel Programming". In: *ACM Sigplan Fortran Forum*. Vol. 17. 2. ACM. 1998, pp. 1–31.

[4] Julien Bourgeois, Vaidy Sunderam, Jaroslaw Slawinski, and Bogdan Cornea. "Extending Executability of Applications on Varied Target Platforms". In: *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE. 2011, pp. 253–260.

[5] Jaroslaw Slawinski, Tiziano Passerini, Umberto Villa, Alessandro Veneziani, and Vaidy Sunderam. "Experiences with Target-Platform Heterogeneity in Clouds, Grids, and On-Premises Resources". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE. 2012, pp. 41–52.

[6] Jaroslaw Slawinski, Umberto Villa, Tiziano Passerini, Alessandro Veneziani, and Vaidy Sunderam. "Issues in Communication Heterogeneity for Message-Passing Concurrent Computing". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 93–102.

[7] Tiziano Passerini, Jaroslaw Slawinski, Umberto Villa, and Vaidy Sunderam. “Experiences with Cost and Utility Tradeoffs on IaaS Clouds, Grids and On-Premise Resources”. In: *International Conference on Cloud Engineering, 2014 IEEE, to appear*. IEEE.

[8] Tiziano Passerini, Jaroslaw Slawinski, Umberto Villa, Alessandro Veneziani, and Vaidy Sunderam. *Experiences with a Computational Fluid Dynamics Code on Clouds, Grids, and On-Premise Resources*. Tech. rep. TR-2012-016. Emory, Mathematics and Computer Science, 2012.

[9] Jaroslaw Slawinski and Vaidy Sunderam. *Approaching Utility Computing via Adaptive Multi-Target Deployment*. Tech. rep. TR-2014-004. Emory, Mathematics and Computer Science, 2012.

[10] Jaroslaw Slawinski and Vaidy Sunderam. *Multi-target Application Deployment with Assistance of Automatic Adaptive Metabuild*. Tech. rep. TR-2014-005. Emory, Mathematics and Computer Science, 2012.

[11] Jaroslaw Slawinski and Vaidy Sunderam. *Multi-Target Deployment for eScience Applications*. Tech. rep. TR-2014-006. Emory, Mathematics and Computer Science, 2012.

[12] Jaroslaw Slawinski and Vaidy Sunderam. “Adapting MPI to MapReduce PaaS Clouds: An Experiment in Cross-Paradigm Execution”. In: *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*. IEEE Computer Society. 2012, pp. 199–203.

[13] Jaroslaw Slawinski and Vaidy Sunderam. “Towards Computing as a Utility via Adaptive Middleware: An Experiment in Cross-paradigm Execution”. In: *Parallel Processing Letters* 23.02 (2013).

[14] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E. Mark, and Herman J. Berendsen. “GROMACS: Fast, Flexible, and Free”. In: *Journal of computational chemistry* 26.16 (2005), pp. 1701–1718. ISSN: 0192-8651. URL: http://dx.doi.org/10.1002/jcc.20291.

[15] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S

Schreiber, et al. “The NAS Parallel Benchmarks”. In: *International Journal of High Performance Computing Applications* 5.3 (1991), pp. 63–73.

[16] Peter Skomoroch. “MPI Cluster Programming with Python and Amazon EC2”. In: *PyCon 2008. Chicago*. 2008.

[17] Jaroslaw Slawinski, Magdalena Slawinska, and Vaidy Sunderam. “Unibus-managed Execution of Scientific Applications on Aggregated Clouds”. In: *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE. 2010, pp. 518–521.

[18] Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki. “Towards Efficient MapReduce Using MPI”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2009), pp. 240–249.

[19] Magdalena Slawinska, Jaroslaw Slawinski, and Vaidy Sunderam. “A Practical, SCVM-based Approach to Enhance Portability and Adaptability of HPC Application Build Systems”. In: *International MultiConference of Engineering and Computer Scientists*. 2012.

[20] Sadaf R Alam, Jeffery A Kuehn, Richard F Barrett, Jeff M Larkin, Mark R Fahey, Ramanan Sankaran, and Patrick H Worley. “Cray XT4: An Early Evaluation for Petascale Scientific Simulation”. In: *Supercomputing, 2007. SC’07. Proceedings of the 2007 ACM/IEEE Conference on*. IEEE. 2007, pp. 1–12.

[21] Yu-Heng Tseng and Chris Ding. “Efficient Parallel I/O in Community Atmosphere Model (CAM)”. In: *International Journal of High Performance Computing Applications* 22.2 (2008), pp. 206–218.

[22] Warren M Washington, Lawrence Buja, and Anthony Craig. “The Computational Future for Climate and Earth System Models: On the Path to Petaflop and Beyond”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 367.1890 (2009), pp. 833–846.

[23] Shahbaz Memon, Morris Riedel, Florian Janetzko, Borries Demeler, Gary Gorbet, Suresh Marru, Andrew Grimshaw, Lahiru Gunathilake, Raminder Singh, Norbert Attig,

et al. “Advancements of the UltraScan scientific gateway for open standards-based cyberinfrastructures”. In: *Concurrency and Computation: Practice and Experience* (2014).

[24] Dan Bonachea and Jason Duell. “Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations”. In: *International Journal of High Performance Computing and Networking* 1.1 (2004), pp. 91–99.

[25] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. “Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI”. In: *Supercomputing, 2007. SC’07. Proceedings of the 2007 ACM/IEEE Conference on*. IEEE. 2007, pp. 1–10.

[26] Geoffroy Vallee, Thomas Naughton, Christian Engelmann, Hong Ong, and Stephen L Scott. “System-Level Virtualization for High Performance Computing”. In: *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*. IEEE. 2008, pp. 636–643.

[27] Ludovico Gardenghi, Michael Goldweber, and Renzo Davoli. “View-OS: A New Unifying Approach Against the Global View Assumption”. In: *Computational Science–ICCS 2008*. Springer, 2008, pp. 287–296.

[28] Marc J. Rochkind. *Advanced UNIX Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985. ISBN: 0-13-011818-4.

[29] Ian Foster and Carl Kesselman. “Globus: A Metacomputing Infrastructure Toolkit”. In: *The International Journal of Supercomputer Applications and High Performance Computing* 11.2 (1997), pp. 115–128.

[30] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. “The Eucalyptus Open-source Cloud-computing System”. In: *9th IEEE International Symposium on Cluster Computing and the Grid, Shanghai, China*. 2009.

[31] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa. “Science Clouds: Early Experiences in Cloud Computing for Scientific Applications”. In: *CCA’08: Cloud Computing and Its Application*. 2008.

[32] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. *AppScale Design and Implementation*. Tech. rep. UCSB, 2009.

[33] Michael A Kozuch, Michael P Ryan, Richard Gass, Steven W Schlosser, David O'Hallaron, James Cipar, Elie Krevat, Julio López, Michael Stroucken, and Gregory R Ganger. "Tashi: Location-Aware Cluster Management". In: *Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM. 2009, pp. 43–48.

[34] Jaroslaw Slawinski, Magdalena Slawinska, and Vaidy Sunderam. "The Unibus Approach to Provisioning Software Applications on Diverse Computing Resources". In: *International Conference On High Performance Computing, 3rd International Workshop on Service Oriented Computing*. 2009.

[35] Hyunjoo Kim, Yaakoub El-Khamra, Shantenu Jha, and Manish Parashar. "An Autonomic Approach to Integrated HPC Grid and Cloud Usage". In: *e-Science, 2009. e-Science'09. Fifth IEEE International Conference on*. IEEE. 2009, pp. 366–373.

[36] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. "High-Performance Cloud Computing: A View of Scientific Applications". In: *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*. IEEE. 2009, pp. 4–16.

[37] J Craig. "Cloud Coalition: rPath, newScale, and Eucalyptus Systems Partner on Self-Service Public and Private Cloud". In: *Enterprise Management Associates* (2010).

[38] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. "Cloud Computing and Grid Computing 360-Degree Compared". In: *Grid Computing Environments Workshop, 2008. GCE'08*. IEEE. 2008, pp. 1–10.

[39] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. "A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing". In: (2010), pp. 115–131.

[40] Constantinos Evangelinos and C Hill. "Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon EC2". In: *ratio* 2.2.40 (2008), pp. 2–34.

[41] Derek Gottfrid. “Self-Service, Prorated Super Computing Fun!” In: *The New York Times* 1 (2007).

[42] Sayaka Akioka and Yoichi Muraoka. “HPC Benchmarks on Amazon EC2”. In: *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*. IEEE. 2010, pp. 1029–1034.

[43] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. “The cost of doing science on the cloud: the montage example”. In: *2008 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2008, pp. 1–12.

[44] Sandro Salsa. “Partial Differential Equations in Action”. In: *From Modelling to Theory, Springer-Verlag, Milan* (2008).

[45] Alessandro Veneziani, Luca Formaggia, and Fausto Saleri. *Solving Numerical PDE's: Problems, Applications, Exercises*. Springer Verlag, 2012.

[46] Alexandre Ern and Jean-Luc Guermond. *Theory and Practice of Finite Elements*. Springer-Verlag, New York, 2004.

[47] Susanne C. Brenner and Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Vol. 15. Springer, 2008.

[48] Luigi Quartapelle. *Numerical Solution of the Incompressible Navier–Stokes Equations*. Vol. 113. Springer, 1993.

[49] Howard C Elman, David J Silvester, and Andrew J Wathen. *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*. Oxford University Press, 2005.

[50] C.R. Ethier and D.A. Steinman. “Exact Fully 3D Navier–Stokes Solutions for Benchmarking”. In: *International Journal for Numerical Methods in Fluids* 19.5 (1994), pp. 369–375.

[51] Christophe Geuzaine and Jean-François Remacle. “Gmsh: a 3-D Finite Element Mesh Generator with Built-in Pre- and Post-Processing Facilities”. In: *International Journal for Numerical Methods in Engineering* 79.11 (2009), pp. 1309–1331.

[52] Preston M Smith. “A Cost-Benefit Analysis of a Campus Computing Grid”. MA thesis. Purdue University, 2011.

[53] Abhinav Bhatele, Todd Gamblin, Steven H. Langer, Peer-Timo Bremer, Erik W. Draeger, Bernd Hamann, Katherine E. Isaacs, Aaditya G. Landge, Joshua A. Levine, Valerio Pascucci, Martin Schulz, and Charles H. Still. “Mapping Applications with Collectives over Sub-Communicators on Torus Networks”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’12. IEEE Computer Society Press, 2012.

[54] J. Hursey, J.M. Squyres, and T. Dontje. “Locality-Aware Parallel Process Mapping for Multi-Core HPC Systems”. In: *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE. 2011, pp. 527–531.

[55] Hari Subramoni, Sreeram Potluri, K Kandalla, B Barth, Jérôme Vienne, Jeff Keasler, K Tomko, K Schulz, Adam Moody, and Dhabaleswar K Panda. “Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 70.

[56] Marina Piccinelli, Alessandro Veneziani, David A Steinman, Andrea Remuzzi, and Luca Antiga. “A Framework for Geometric Analysis of Vascular Structures: Application to Cerebral Aneurysms”. In: *Medical Imaging, IEEE Transactions on* 28.8 (2009), pp. 1141–1155.

[57] Luca Formaggia, Karl Perktold, and Alfio Quarteroni. “Basic Mathematical Models and Motivations”. In: *Cardiovascular mathematics*. Springer, 2009, pp. 47–75.

[58] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Text in Applied Mathematics. New York: Springer, 2007.

[59] D. M. Sforza, C. M. Putman, and J. R. Cebral. “Hemodynamics of Cerebral Aneurysms”. In: *Annu. Rev. Fluid Mech.* 41 (2009), pp. 91–107.

[60] T. Passerini, L. M. Sangalli, S. Vantini, M. Piccinelli, S. Bacigaluppi, L. Antiga, E. Boccardi, P. Secchi, and A. Veneziani. "An Integrated Statistical Investigation of Internal Carotid Arteries of Patients affected by Cerebral Aneurysms". In: *CVET* 3 (1 2012), pp. 26–40.

[61] David A Steinman, Yiemeng Hoi, Paul Fahy, Liam Morris, Michael T Walsh, Nicolas Aristokleous, Andreas S Anayiotos, Yannis Papaharilaou, Amirhossein Arzani, Shawn C Shadden, et al. "Variability of Computational Fluid Dynamics Solutions for Pressure and Flow in a Giant Aneurysm: the ASME 2012 Summer Bioengineering Conference CFD Challenge". In: *Journal of Biomechanical Engineering* 135.2 (2013).

[62] R. S. Tuminaro and T. Tong. "Parallel Smoothed Aggregation Multigrid: Aggregation Strategies on Massively Parallel Machines". In: *SC Conference* (2000).

[63] S.S. Shende and A.D. Malony. "The TAU Parallel Performance System". In: *International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311.

[64] Katherine Yelick, Susan Coghlan, Brent Draney, Richard Shane Canon, et al. "The Magellan Report on Cloud Computing for Science". In: *US Department of Energy, Washington DC, USA, Tech. Rep* (2011).

[65] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven HM Buijssen, Matthias Grajewski, and Stefan Turek. "Exploring Weak Scalability for FEM Calculations on a GPU-Enhanced Cluster". In: *Parallel Computing* 33.10 (2007), pp. 685–699.

[66] Christina Hoffa, Gaurang Mehta, Timothy Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. "On the Use of Cloud Computing for Scientific Workflows". In: *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. IEEE. 2008, pp. 640–645.

[67] Cynthia B Lee and Allan E Snavely. "Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers". In: *Proceedings of the 16th international symposium on High performance distributed computing*. ACM. 2007, pp. 107–116.

[68] Giorgos Cheliotis, Chris Kenyon, Rajkumar Buyya, and A Melbourne. "Grid Economics: 10 Lessons from Finance". In: *GRIDS Lab and IBM Research Zurich, Melbourne, Tech. Rep* (2003).

[69] B.N. Chun and D.E. Culler. "User-centric performance analysis of market-based cluster batch schedulers". In: *2002 2nd International Symposium on Cluster Computing and the Grid*. IEEE/ACM. 2002, pp. 30–30.

[70] Joao Nuno Silva, Paulo Ferreira, and Luís Veiga. "Service and Resource Discovery in Cycle-Sharing Environments with a Utility Algebra". In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–11.

[71] José Simão and Luís Veiga. "QoE-JVM: An Adaptive and Resource-Aware Java Runtime for Cloud Computing". In: (2012), pp. 566–583.

[72] David E Irwin, Laura E Grit, and Jeffrey S Chase. "Balancing Risk and Reward in a Market-Based Task Service". In: *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*. IEEE. 2004, pp. 160–169.

[73] John J Rehr, Fernando D Vila, Jeffrey P Gardner, Lucas Svec, and Micah Prange. "Scientific Computing in the Cloud". In: *Computing in Science & Engineering* 12.3 (2010), pp. 34–43.

[74] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. 2010, pp. 159–168.

[75] Guohui Wang and TS Eugene Ng. "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center". In: *INFOCOM, 2010 Proceedings IEEE*. IEEE. 2010, pp. 1–9.

[76] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick HJ Epema. "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing". In: *Parallel and Distributed Systems, IEEE Transactions on* 22.6 (2011), pp. 931–945.

[77] Doug Howe, Maria Costanzo, Petra Fey, Takashi Gojobori, Linda Hannick, Winston Hide, David P Hill, Renate Kania, Mary Schaeffer, Susan St Pierre, et al. "Big Data: The Future of Biocuration". In: *Nature* 455.7209 (2008), pp. 47–50.

[78] Morris A Bender, Isaac Ginis, Robert Tuleya, Biju Thomas, and Timothy Marchok. "The Operational GFDL Coupled Hurricane–Ocean Prediction System and a Summary of Its Performance." In: *Monthly Weather Review* 135.12 (2007).

[79] SH Anijdan, A Bahrami, HR Hosseini, and A Shafyei. "Using Genetic Algorithm and Artificial Neural Network Analyses to Design an Al–Si Casting Alloy of Minimum Porosity". In: *Materials & design* 27.7 (2006), pp. 605–609.

[80] Marta Łuksza and Michael Lässig. "A predictive fitness model for influenza". In: *Nature* 507.7490 (2014), pp. 57–61.

[81] Magdalena Slawinska, Jaroslaw Slawinski, and Vaidy Sunderam. "Portable Builds of HPC Applications on Diverse Target Platforms". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–8.

[82] Magdalena Slawinska, Jaroslaw Slawinski, Dawid Kurzyniec, and Vaidy Sunderam. "Enhancing Portability of HPC Applications across High-end Computing Platforms". In: *Parallel and Distributed Processing Symposium, International* 0 (2007), p. 143.

[83] Kenneth Hoste, Jens Timmerman, Andy Georges, and Stijn De Weirdt. "EasyBuild: Building Software with Ease". In: *High Performance Computing, Networking Storage and Analysis, SC Companion:* (2012), pp. 572–582.

[84] Philip J Guo. "CDE: Run Any Linux Application On-demand Without Installation". In: *USENIX Large Installation System Administration (LISA)* (2011).

[85] R. Clint Whaley and Jack J. Dongarra. "Automatically Tuned Linear Algebra Software". In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society. 1998, pp. 1–27.

[86] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[87] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. “A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard”. In: *Parallel computing* 22.6 (1996), pp. 789–828.

[88] Stephen J. Chapman. *MATLAB Programming for Engineers*. Thomson Engineering, 2008. ISBN: 049524449X.

[89] Steven J Plimpton and Karen D Devine. “MapReduce in MPI for Large-Scale Graph Algorithms”. In: *Parallel Computing* 37.9 (2011), pp. 610–632.

[90] Leslie G. Valiant. “A Bridging Model for Multi-core Computing”. In: *Journal of Computer and System Sciences* 77.1 (2011).

[91] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Publications Co., 2011.

[92] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. “Twister: A Runtime for Iterative Mapreduce”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM. 2010.

[93] Grzegorz Malewicz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 international conference on Management of data*. ACM. 2010, pp. 135–146.

[94] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. “CloudBLAST: Combining Mapreduce and Virtualization on Distributed Resources for Bioinformatics Applications”. In: *eScience, 2008. eScience’08. IEEE Fourth International Conference on*. IEEE. 2008, pp. 222–229.

[95] J. Ansel, K. Arya, and G. Cooperman. “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop”. In: *Parallel & Dist. Proc., 2009. IPDPS 2009. IEEE Int. Symposium on*. IEEE. 2009.

[96] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. “Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System”. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE. 2009, pp. 198–207.

[97] Jonathan Hill and David B Skillicorn. “Lessons Learned from Implementing BSP”. In: *Future generation computer systems* 13.4 (1998), pp. 327–335.

Bibliography: Electronic Resources

[98] *Amazon Elastic Compute Cloud (Amazon EC2)*. aws.amazon.com/ec2. 2014.

[99] Jaroslaw Slawinski and Vaidy Sunderam. *Towards Semi-Automatic Deployment of Scientific and Engineering Applications*. http://dx.doi.org/10.6084/m9.figshare.791570. ACM/IEEE Supercomputing Conference, 2013 SC'13. First Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE1). 2013.

[100] *Amazon Elastic MapReduce (Amazon EMR)*. aws.amazon.com/elasticmapreduce. 2014.

[101] *RightScale – Cloud Management Platform*. www.rightscale.com. 2014.

[102] *Docker*. www.docker.io. 2014.

[103] *Filesystem in Userspace*. fuse.sourceforge.net. 2014.

[104] *The Simple Cloud API – Writing portable, interoperable applications for the cloud*. www.ibm.com/developerworks/opensource/library/os-simplecloud. 2014.

[105] *Open Cloud Manifesto*. www.opencloudmanifesto.org. 2011.

[106] *Rackspace – the open cloud company*. www.rackspace.com. 2014.

[107] *PortableApps.com*. portableapps.com. 2014.

[108] *LifeV Project*. www.lifev.org. 2014.

[109] *CAELinux open-source powered engineering*. www.caelinux.com/CMS. 2014.

[110] *Code_Aster*. www.code-aster.org. 2014.

[111] *Code_Saturne*. research.edf.com/research-and-the-scientific-community/software/code-saturne/introduction-code-saturne-80058.html. 2014.

[112] *Salome – The Open Source Integration Platform for Numerical Simulation*. www.salome-platform.org. 2014.

[113] *OpenFOAM – The open source CFD toolbox*. www.openfoam.com. 2014.

[114] *Elmer – Open Source Finite Element Software for Multiphysical Problems*. www.csc.fi/english/pages/elmer. 2014.

[115] *Using OpenFOAM with Amazon EC2*. www.openfoam.com/resources/ec2.php. 2012.

[116] Schberl Joachim, Gerstmayr Hannes, and Gaisbauer Robert. *NETGEN – automatic mesh generator*. www.hpfem.jku.at/netgen. 2014.

[117] George Karypis and Vipin Kumar. *Family of Graph and Hypergraph Partitioning Software*. glaros.dtc.umn.edu/gkhome/views/metis. University of Minnesota, Minneapolis, MN, 2014.

[118] Sandia National Laboratories. *The Trilinos Project*. trilinos.sandia.gov. 2014.

[119] *ParaView*. www.paraview.org. 2014.

[120] *SuiteSparse*. www.cise.ufl.edu/research/sparse/SuiteSparse. 2014.

[121] David M. Mount and Sunil Arya. *ANN: Library for Approximate Nearest Neighbour Searching*. www.cs.umd.edu/~mount/ANN. 2014.

[122] *Boost C++ Libraries*. www.boost.org. 2014.

[123] The HDF Group. *Hierarchical data format version 5*. www.hdfgroup.org/HDF5. 2014.

[124] *TOP500 Supercomputer Sites*. top500.org. 2014.

[125] *Amazon Web Services/Developer Tools*. aws.amazon.com/developertools. 2014.

[126] *boto: A Python interface to Amazon Web Services*. docs.pythonboto.org/en/latest. 2014.

[127] MIT. *StarCluster*. web.mit.edu/stardev/cluster. 2014.

[128] AMD. *AMD Core Math Library (ACML)*. developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml. 2014.

[129] Intel. *Intel Math Kernel Library (Intel MKL)*. software.intel.com/intel-mkl. 2014.

[130] *Performance Analysis and Visualization at Exascale (PAVE)*. computation-rnd.llnl.gov/performance-analysis-through-visualization. 2014.

[131] *OpenMPI 1.6.4 manual*. www.open-mpi.org/doc/v1.6/man1/mpirun.1.php. 2014.

[132] *The Laboratory for Modeling and Scientific Computing MOX. The Aneurisk project.* mox.polimi.it/it/progetti/aneurisk. 2014.

[133] *AneuriskWeb – The Aneurisk dataset repository*. ecm2.mathcs.emory.edu/aneuriskweb. 2014.

[134] François Pellegrini. *Scotch and libScotch 6.0 Users Guide*. gforge.inria.fr/docman/view.php/248/8260/scotch_user6.0.pdf. 2014.

[135] *Extreme Science and Engineering Discovery Environment Project*. www.xsede.org. 2014.

[136] *Penguin Computing On Demand / Indiana University*. podiu.penguincomputing.com. 2012.

[137] *HPC Cloud Service, Penguin Computing*. www.penguincomputing.com/Services/HPCCloud. 2014.

[138] *GNU Make*. www.gnu.org/software/make. 2014.

[139] *CMake*. www.cmake.org. 2014.

[140] *The Apache Ant Project*. ant.apache.org. 2014.

[141] *SCons*. www.scons.org. 2014.

[142] *Gradle*. www.gradle.org. 2014.

[143] *Autoconf – GNU Project*. www.gnu.org/software/autoconf. 2014.

[144] *The Environment Modules Project*. modules.sourceforge.net. 2014.

[145] *The RPM Package Manager (RPM)*. rpm.org. 2014.

[146] *Deployment by ADAPT*. code.google.com/p/dadapt. 2014.

[147] *The VisIt Project*. https://wci.llnl.gov/codes/visit. 2014.

[148] *OCCI – Open Cloud Computing Interface*. occi-wg.org. 2014.

[149] *OpenNebula – Flexible Enterprise Cloud Made Simple*. www.opennebula.org. 2014.

[150] *Eucalyptus – Open Source AWS Compatible Private Clouds*. www.eucalyptus.com/eucalyptus-cloud. 2014.

[151] *Apache Hadoop*. hadoop.apache.org. 2014.

[152] *Apache Hama*. hama.apache.org. 2014.

[153] *MPICH – High-Performance Portable MPI*. www.mpich.org. 2014.