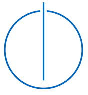


Fakultät für Informatik
Technische Universität München

TUM

Technische Universität München

Fakultät für Informatik

Lehrstuhl für Informatik mit Schwerpunkt
Wissenschaftliches Rechnen

Resource-Aware and Elastic Parallel Software Development for Distributed-Memory HPC Systems

Ao Mo-Hellenbrand

Vollständiger Abdruck der von der Fakultät für Informatik der Technische Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

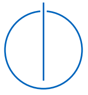
Vorsitzender:

Prof. Bernd Brügge, Ph.D.

Prüfende der Dissertation:

1. Prof. Dr. Hans-Joachim Bungartz
2. Prof. Dr. Martin Schulz

Die Dissertation wurde am 15.01.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 26.03.2019 angenommen.



Fakultät für Informatik
Technische Universität München

TUM

Technical University of Munich

Department of Computer Science

Chair of Scientific Computing in Computer Science

Resource-Aware and Elastic Parallel Software Development for Distributed-Memory HPC Systems

Ao Mo-Hellenbrand

Full imprint of the dissertation approved by the Department of Computer Science of Technical University of Munich to obtain the academic degree of

Doctor of Natural Sciences (Dr. rer. nat.).

Chairman:

Prof. Dr. Bernd Brügge

Examiners of the dissertation:

1. Prof. Dr. Hans-Joachim Bungartz
2. Prof. Dr. Martin Schulz

The dissertation was submitted to the Technical University of Munich on 15.01.2019 and was accepted by the Department of Computer Science on 26.03.2019.

Abstract

As the exascale computing era is soon be to expected, computational resource and energy efficiency have become an important theme in HPC research. Indeed, most of the current HPC challenges such as resource/energy inefficiency are rooted from the incompatibility of the traditional static resource management and programming model with the dynamic behavior of modern HPC applications as well as the increasing computing power and complexity in HPC systems. Motivated to overcome such challenges, we propose a solution that aims to achieve optimal system throughput or resource/energy efficiency via a flexible resource management infrastructure and a resource-elastic programming framework – Elastic MPI. This project is part of the *Transregional Collaborative Research Center 89: Invasive Computing* research effort.

Realizing resource awareness and elasticity in HPC requires support from both systems and applications, i.e., HPC systems should support dynamic resource allocation and resource-elastic execution, and parallel applications should have the ability to adapt their execution at runtime to fit different amounts of resources. Efficient malleable applications that can quickly adapt to resource changes are the keys to eliminating idling and inefficiently utilized resources in the system. The major challenges of introducing resource adaptivity to HPC applications are that it requires flexible data decomposition and load balancing schemes, and that it might introduce significant overhead due to frequent data redistribution and migration.

This work presents our pilot study in malleable parallel software development for distributed-memory HPC systems using the Elastic MPI programming framework. We discuss elastic programming models for different application types, including those based on the SPMD or the master-worker execution model as well as those with single or multiple resource-elastic computational phases. We implemented three malleable applications ranging from the classical grid-based simulations to the embarrassingly parallel problems. To assess the impacts of runtime resource adaptivity, we conduct performance analysis on each application individually to measure the resource adaptation overhead as well as comparing their runtime and resource efficiency with those of their static MPI counterparts.

Contents

Abstract v

Contents vii

List of Algorithms ix

List of Figures xi

List of Tables xiii

Glossary xv

I Introduction 1

1 Introduction 3
- 1.1 Current Challenges in HPC . . . 3
- 1.2 Resource Awareness and Elasticity as a Solution . . . 5
- 1.3 Invasive Computing . . . 5
- 1.4 Elastic MPI . . . 7
- 1.5 Contribution of the Current Work . . . 8
- 1.6 Outline . . . 8

2 Related Work 11
- 2.1 Dynamic Processes Support by the MPI Standard . . . 11
- 2.2 MPI Sessions . . . 13
- 2.3 User-level Fault Tolerance in MPI . . . 13
- 2.4 Charm++ and Adaptive MPI . . . 14
- 2.5 High Performance ParallelX (HPX) . . . 15
- 2.6 Invasive X10 (*i*X10) . . . 16
- 2.7 SALSA and PCM Extensions with IOS . . . 17
- 2.8 Parallel Virtual Machine (PVM) . . . 17
- 2.9 Cloud Computing . . . 18
- 2.10 Summary . . . 19

II Elastic MPI Framework 21

3 Elastic MPI Infrastructure 23
- 3.1 Overview . . . 23

3.2 Elastic MPI Library 25
3.3 Elastic Resource Manager 29
3.4 Limitations and Known Issues 36
3.5 Summary 38

4 Parallel Programming with Elastic MPI **39**
4.1 Classification of HPC Applications 39
4.2 SPMD with Single Computation Phase 40
4.3 SPMD with Multiple Computation Phases 44
4.4 Master-Worker with Single Computation Phase 44
4.5 Master-Worker with Multiple Computation Phases 49
4.6 Summary 51

III Resource-Aware and Elastic Parallel Software Development **53**

5 Elastic Parallel Tsunami Simulation with Adaptive Mesh Refinement **55**
5.1 Sierpiński Space-filling Curves 56
5.2 Tsunami Simulation in sam(oa)^2 58
5.3 Resource-elastic Transformation 64
5.4 Performance Evaluation 66
5.5 Summary 72

6 Elastic Parallel Oil Reservoir Simulation with Adaptive Mesh Refinement **75**
6.1 The SPE10 Benchmark Simulation Scenario 76
6.2 Porous Media Flow Simulation in sam(oa)^2 78
6.3 Resource-elastic Transformation 85
6.4 Performance Evaluation 89
6.5 Summary 96

7 Statistical Inverse Problem Solver with Elastic Parallel Surrogate Construction **99**
7.1 Statistical Inverse Problems 99
7.2 Surrogate Model Construction with Sparse Grids 106
7.3 Case Study: Inference of Obstacle Locations in Laminar Flow 111
7.4 Performance Evaluation 118
7.5 Summary 124

IV Conclusion **127**

8 Conclusion and Outlook **129**
8.1 Conclusion 129
8.2 Outlook 131

Bibliography **133**

List of Algorithms

4.1 MPI program: SPMD Single Phase 41
4.2 Elastic MPI program: SPMD Single Phase 42
4.3 Elastic MPI program: SPMD Multiple Phases 45
4.4 MPI program: M-W Single Phase 46
4.5 Elastic MPI program: M-W Single Phase (Main function) 47
4.6 Elastic MPI program: M-W Single Phase (Master & Worker functions) . . . 48
4.7 Elastic MPI program: M-W Multiple Phases (Main function) 50
5.1 Main algorithm of the parallel tsunami simulation from sam(oa)^2 63
5.2 Main algorithm of the Elastic MPI tsunami simulation 65
5.3 Resource adaptation function of the Elastic MPI tsunami simulation 66
6.1 Main algorithm of the oil reservoir simulation from sam(oa)^2 86
6.2 Main algorithm of the Elastic MPI oil reservoir simulation 87
6.3 Resource adaptation of the Elastic MPI oil reservoir simulation 88
7.1 Metropolis-Hastings Algorithm 104
7.2 Parallel Tempering Algorithm 105
7.3 Main algorithm for locating obstacles in a fluid channel 116
7.4 Master and Worker functions for surrogate construction 117

List of Figures

1.1 Abstract overview of subprojects in *Invasive Computing* 6

3.1 Overview of the Elastic MPI System 24
3.3 SLURM overview 32
3.4 SLURM-based Elastic MPI infrastructure overview 33

5.1 Creation of adaptive mesh with Sierpiński curve 57
5.2 Demonstration of streams- and stacks-based data access scheme 58
5.3 Demonstration of parallelization and load balancing 61
5.4 Benchmark simulation of the Tohoku tsunami 69
5.5 Resource change profile during a test run 70
5.6 Tsunami simulation performance analysis 71

6.1 Permeability field of the simulation domain 76
6.2 Full-sized SPE10 simulation with 85 layers 77
6.3 2.5-D adaptive prismatic grid in sam(oa)^2 77
6.4 Reduced SPE10 simulation with 16 layers 91
6.5 Resource and workload profile of Elastic MPI test with random scheduler . 92
6.6 Average execution time of one step vs. number of processes 94
6.7 Resource and workload profile of Elastic MPI test with stepping scheduler . 95
6.8 Resource profile vs. execution time and CPU hours of three test runs 95

7.1 1-D function interpolation on unit interval 107
7.2 A 2-D full grid, hierarchical decomposition and sparse grid 108
7.3 2-D adaptive sparse grid 110
7.4 A 2-D fluid channel with four obstacles at unknown locations 112
7.5 Staggered grid 113
7.6 A forward simulation of 2-D fluid channel with four obstacles 119
7.7 Resource profile of one test run of the obstacle location inverse problem . . 120
7.8 Most probable obstacle locations 123
7.9 Histogram of the MCMC output samples per dimension 123
7.10 Resource profile vs. execution time and CPU hours of two test runs 124

List of Tables

5.1 The SuperMUC CPU cluster specifications 67
5.2 Average execution time of elastic MPI functions 70
5.3 Average exec. time percentage of computational tasks 70
5.4 Comparison of execution time and CPU hours 70

6.1 Specifications of the VM cluster and its host machine 89
6.2 Execution time of Elastic MPI functions 93
6.3 Execution time of computational tasks 93
6.4 Execution time and CPU hours of three test runs 96
6.5 Execution time distribution of three test runs 96

7.1 Comparison of number of grid points of $l = 5$ full grids and sparse grids . . 109
7.2 Execution time of each phase 121
7.3 Execution time of Elastic MPI functions 121
7.4 Computational efforts comparison 121
7.5 Execution time and CPU hours of two test runs 124

8.1 Characteristics summary of Elastic MPI applications 129

Glossary

Definition and explanation of important terms used in this thesis are listed below.

Glossary

communication-intensive
A parallel application is said to be communication-intensive if there are non-trivial data dependency and requirements for communication and synchronization among its subtasks.

compute-bound / CPU-bound
An application is compute-bound or CPU-bound when its progress is limited by the processor speed.

Elastic MPI
An infrastructure and programming framework providing resource awareness and elasticity for distributed-memory HPC systems.

embarrassingly parallel
A parallel application is considered embarrassingly parallel if there is little to none data dependency or requirement for communication among its subtasks.

Invasive Computing
A DFG funded research project focuses on the development of resource-aware programming for future parallel computing systems.

invasive
Being resource-aware and elastic.

communication-bound / bandwidth-bound
An application is communication-bound or bandwidth-bound when its progress is limited by the networking due to intensive communication among the executing processes.

malleable
Able to adapt the execution to fit different amount of resources.
This term is interchangeable with *resource-elastic* for applications.

process
An executing instance of a computer program with a private memory space. It is unaware of information located outside of its own memory and requires explicit communication in order to obtain such information.

resource adaptation window
The period between the `MPI_Comm_adapt_begin` and `MPI_Comm_adapt_commit` functions. It is a time window in which all resource adaptation-related operations should be performed.

resource-aware
Knowing the amount and types of resources that can be utilized and be sensible to their real-time changes.

resource-elastic
For systems, it means having support for dynamic resource allocation and resource-elastic execution. For applications, it means able to adapt the execution to fit different amount of resources. This term is interchangeable with *malleable* for applications.

resource-static
Having no change in regards of resources.

SuperMUC
The supercomputer of the Leibniz Supercomputing Centre (Leibniz-Rechenzentrum, LRZ) in Munich, Germany.

thread
An executing instance of a computer program with a shared memory space. Its information is shared with other threads that have access to the same shared memory space, therefore, explicit communication between threads is not required for exchange of information.

Acronyms

ALU Arithmetic Logic Unit

AMR Adaptive Mesh Refinement

API Application Programming Interface

CPU Central Processing Unit

DFG German Research Foundation (Deutsche Forschungsgemeinschaft)

flops Floating Point Operations Per Second

FPGA Field Programmable Gate Array

GPGPU General Purpose GPU

GPU Graphics Processing Unit

HPC High Performance Computing

LRZ Leibniz Supercomputing Centre (Leibniz-Rechenzentrum)

MPI Message Passing Interface

NUMA Non-Uniform Memory Access

PDE Partial Differential Equation

PGAS Partitioned Global Address Space

PMI Process Management Interface

SFC Space-filling Curve

SLURM Simple Linux Utility for Resource Management

SPMD Single Program Multiple Data

SWE Shallow Water Equations

TCPA Tightly-Coupled Processor Array

PART I

INTRODUCTION

1 Introduction

High Performance Computing (HPC) refers to the practice of aggregating computing power to solve complex problems efficiently. For more than three decades, HPC systems have been prevailing platforms in the fields of science and engineering. They are tailored for large applications that are data-intensive and require a lot of computational power and resources. In recent years, however, technological advancements in both computer hardware and software development have posed several challenges in HPC. Research has been carried out in search of remedies for these challenges. Many approaches have been attempted. In this work, we attempt to find a solution that can resolve some urgent HPC problems with canonical tools and a novel programming paradigm.

1.1 Current Challenges in HPC

The current trend in computer hardware development tends towards increasing the amount of computational resources and the degree of heterogeneity. It is likely that thousands of processing units will be integrated on a single chip device in the imminent future [1]. With the performance of current supercomputers measured in petaflops (10^{15} Floating Point Operations Per Second), we are expected to soon enter the exascale (exaflops, or 10^{18} flops) era. In addition to the growth in size, modern HPC systems also grow in complexity as they include a wider variety of architectures, such as many-core CPUs, GPUs, GPGPUs, FPGAs[1] and other special purpose accelerators. The number of processing units and the heterogeneity make it hard to achieve optimal resource usage with traditional programming models, as they are not designed to handle such complexity.

Challenges also arise from HPC software development. Many modern parallel applications are dynamic, i.e., their computational workload depends on the input and runtime. An example would be a simulation with adaptive mesh refinement (AMR), whose workload constantly changes due to its underlying grid being refined at runtime, which in turn depends on intermediate simulation outputs. For such dynamic applications, static resource allocation causes suboptimal resource utilization due to load imbalances and frequently changing resource requirements. With static resource allocation, applications with multiple phases that have different scalability, as well as applications with limited information on its parallel performance can also suffer from resource inefficiency.

[1] For unexplained abbreviations refer to the Glossary list.

The predominant programming model for distributed-memory system is Message Passing. Message Passing Interface (MPI) [2, 3, 4], a standardized programming interface for Message Passing, was first released in 1994 and has since played an important role in HPC. The programming model behind MPI is mostly based on the configuration that resources do not change during program execution, which is in accordance with the static resource management scheme implemented on most HPC systems. Present-day HPC systems typically operate in space sharing mode [5] in combination with static resource management, which gives parallel tasks exclusive, constant access to the requested resources for their entire execution period. Based on these premises, MPI applications are normally designed and optimized for a fixed set of resources.

While this resource-static management and programming approach provides a very stable and predictable execution environment, it has several shortcomings such as

- resource utilization inefficiency,
- no support for fault tolerance,
- increased difficulties and complexity in code development,
- low compatibility and portability across systems,

and more. These deficiencies have become more prominent with recent development in HPC hardware and software.

Problems of Resource-Static Models for Applications

In the batch environment of an HPC system, task execution is only possible after resource allocation. The *turnaround time* of a task refers to the period from the task being submitted to it being completed, which includes the *wait time* for obtaining the requested resources and the actual *execution time*, i.e.,

$$T_{\text{turnaround}} = T_{\text{wait}} + T_{\text{exec}}.$$

T_{wait} is dependent on the availability of resources, which is in turn affected by several factors such as job priority, current workload on the system, and most importantly, the amount of resources requested by the task. The smaller the request, the quicker it can be fulfilled such that task execution can be started. With the resource-static model, applications are designed and optimized for a fixed resource amount, which means the amount of requested resources cannot be manipulated arbitrarily to reduce T_{wait}.

Users of HPC systems usually have a limited computational budget that is in general measured in resource usage, e.g., CPU-hours. Resource-static models cause suboptimal resource utilization for dynamic applications, applications with multiple phases that have different scalability, and applications with limited information on its parallel performance. Resource utilization inefficiency leads to higher computational costs (CPU hours) for running the application and longer T_{exec}.

Problems of Resource-Static Models for HPC Systems

With the resource-static model, idling resources can be created during resource accumulation for a task, i.e., some resources that are available cannot be utilized until an adequate amount is accumulated to fulfill a task's resource request. Idling resources can also be

generated during task execution due to load imbalances and changes of resource requirements. Since idling and inefficiently utilized sources operate on different power levels, resource inefficiency causes power fluctuations that are inherently dangerous to the system hardware.

Resource inefficiency is economically unfriendly. Performance of large HPC systems such as supercomputers are currently measured in petaflops scale. Powering such a system requires megawatts of electricity, which translates into an annual operating cost of millions of US Dollars. A popular metric for measuring energy efficiency is the performance-to-power ratio, i.e., petaflops per megawatt. Resource inefficiency degrades system performance, which could result in energy waste worth many million dollars. Therefore, minimizing resource inefficiency and maximizing system performance are imperative tasks in the HPC world.

1.2 Resource Awareness and Elasticity as a Solution

Most of the current HPC problems are caused by the incompatibility of the traditional static resource management and programming model with the dynamic behavior of modern HPC applications as well as the increasing size and complexity of HPC systems. Resource awareness and elasticity arise as natural solutions.

Resource awareness refers to the ability to identify the amount and types of resources that can be utilized and being sensible to their real-time changes. Resource elasticity is the ability to adapt the execution to fit the amount of allocated resources that may vary at runtime. Both terms can be applied to systems and applications. For a system, resource-elastic means it supports dynamic resource allocation and resource-elastic execution.

As a general concept, we would like systems and applications to quickly react to the real-time changes of computational workload and resource availability. Resource awareness allows them to sense changes, while resource elasticity gives them the ability to react. Theoretically, this solution can maximize resource utilization, and thus, resolves those problems induced by resource usage inefficiency. To a certain extent, it also provide a remedy for fault tolerance, since systems and applications can sense or detect faulty hardware and adapt accordingly.

1.3 Invasive Computing

The *Transregional Collaborative Research Center 89: Invasive Computing* [6, 1] is a research project funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG). It focuses on the investigation and development of resource-aware programming for future parallel computing systems. It has a wide research scope ranging from embedded devices to large HPC systems in terms of computer architecture. It also covers multiple areas from hardware, system software to applications. The term *invasive* refers to being resource-aware and elastic.

The *Invasive Computing* project first started in 2010. After the completion of two successful funding phases of four years each, it has been extended with a third phase until 2022. The project is a collaboration between three institutions – Friedrich-Alexander

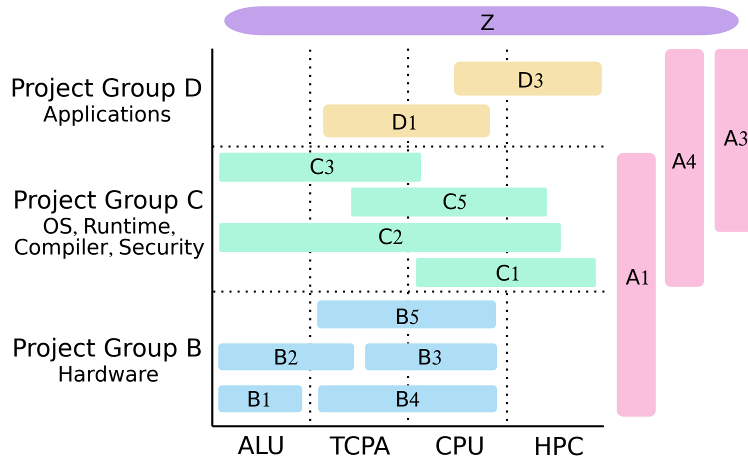


Figure 1.1: Abstract overview of subprojects in *Invasive Computing*

University Erlangen-Nürnberg (FAU), Karlsruhe Institute of Technology (KIT) and the Technical University of Munich (TUM).

Invasive Computing is organized in multiple smaller research groups called subprojects, each of which investigates a certain functional area with the focus on a specific component. Subprojects are clustered in five groups under the letters A, B, C, D and Z. Figure 1.1 provides an abstract overview of all the subprojects.

The horizontal axis enumerates the hardware abstractions including Arithmetic Logic Unit (ALU), Tightly-Coupled Processor Array (TCPA), CPU and HPC, of which TCPA is a novel invention by *Invasive Computing* . The vertical axis enumerates three development abstraction layers: hardware, system software (operating system, runtime, compiler), and applications. Subproject groups B, C and D mount to each of these layers respectively. Subproject groups A and Z reside outside of the plane spanned by the axes, because they do not belong to specific areas due to some overall governing or aggregation properties. The following list provides a summary of the research contents of each subproject group[2]:

- **Group A:** Subproject A1 focuses on resource-aware programming language and framework development. A3 develops efficient scheduling and load balancing methods. And A4 investigates in application pattern characterization, which helps with runtime scheduling decisions.
- **Group B:** Subprojects B1 through B5 cover the hardware level topics of invasive microarchitectures, invasive TCPAs, invasive general-purpose cores (*i*Cores), monitoring and power management, and networks on-chip.
- **Group C:** Subprojects C1, C2, C3 and C5 cover the invasive support for runtime and operating systems, code compilation and generation, as well as system securities.
- **Group D:** Subproject D1 develops robotic applications. And D3 focuses on invasive infrastructure and software development for HPC.
- **Group Z:** Subprojects in the Z group cover general administrative tasks, and coordinate the aggregation and validation of research results from other subprojects.

[2] Subproject numbers are not necessarily continuous due to discontinuation of certain subprojects.

The present work is part of subproject D3: Invasive Computing and HPC, which focuses on extending and applying the Invasive computing concepts to the HPC realm. Section 1.4 elaborates on our research activities, and Section 1.5 illustrates the contributions of this work in detail.

1.4 Elastic MPI: Realization of Resource Awareness and Elasticity on Distributed-Memory HPC Systems

Replacing the conventional static resource management and programming model has become a compelling task in HPC due to recent advancements in technology, such as increase in the amount of computing units and heterogeneity in hardware and increasing dynamic workload behaviors in modern software. Motivated to overcome the some of the HPC challenges, and as part of the *Invasive Computing* research effort, we propose a solution that aims to optimize system performance by realizing resource awareness and elasticity on HPC systems. Indeed, an earlier effort targeted for shared-memory architectures has been successful: a thread-based elastic infrastructure named *iOMP* [7, 8] was implemented, and applications developed upon it have demonstrated improved performance [9, 10, 11].

To extend beyond shared-memory architectures, we have been focusing on implementing resource awareness and elasticity on distributed-memory HPC systems. Our Message Passing-based solution, named *Elastic MPI*, consists of two major components:

- Elastic MPI infrastructure, and
- Elastic MPI applications.

The Elastic MPI infrastructure, implemented by Comprés [12], is the first pillar of our solution. It supports both normal (resource-static) and malleable (resource-elastic) parallel tasks. It consists of a resource manager and a communication library. The resource manager oversees all resources as well as waiting and running tasks. It monitors parallel performance of the running malleable tasks and estimates their scalability. It decides periodically how to reallocate resources for running tasks and launching new tasks, with the goal of optimizing system throughput or energy efficiency. The communication library allows malleable tasks to communicate with the resource manager and reacts on its resource change decisions.

The second pillar of our solution is malleable applications implemented upon the Elastic MPI infrastructure. Developing these applications is the major contribution of the current work. Efficient malleable applications that can quickly adapt to resource changes is the key to eliminating idling and inefficiently utilized resources in the system.

Elastic MPI is designed for general HPC systems. The infrastructure can be installed on conventional Linux clusters with GNU compiler support, transforming the cluster into a resource-elastic environment. The resource manager is an extension to SLURM [13], a popular open-source cluster management and job scheduling system that is widely used among the TOP500 supercomputers. The communication library is an extension of MPICH [14], a popular open-source implementation of the MPI standard. Like MPI, it provides interfaces for both C/C++ and Fortran languages. Normal MPI programs written in C/C++ or Fortran can be transformed into malleable applications using the Elastic MPI library.

1.5 Contribution of the Current Work

This dissertation contributes to the second of the two components in the Elastic MPI solution: developing and analyzing Elastic MPI applications. This is our pilot study in malleable parallel software development for distributed-memory HPC systems.

We abstract Elastic MPI programming models for the Single Program Multiple Data (SPMD) and master-worker execution models, which are usually implemented for communication-intensive and embarrassingly parallel applications respectively. We also discuss how to handle applications with single or multiple resource-elastic computational phases.

With the goal of covering a variety of applications, we implement three malleable software of different parallel characteristics.

- The first application is a *tsunami simulation* implemented with Adaptive Mesh Refinement (AMR). This is a classical grid-based HPC application implementing a SPMD model. It is compute-bound (CPU-bound) and has a very dynamic computational workload.
- The second application is a *oil reservoir simulation*, which is also a classical grid-based HPC application and has a very similar computational workflow as the tsunami simulation. The differences are that it is communication-bound (bandwidth-bound) and has a static workload.
- The third application is a *statistical inverse problem solver with surrogate construction*. This is an embarrassingly parallel application implementing a master-worker model. It has multiple resource-elastic phases and a different workload for each phase.

For each application, we first discuss its theory and malleable implementation with Elastic MPI, then we conduct a series of performance tests to analyze the impact of introducing runtime resource adaptivity, e.g., the overhead from Elastic MPI function calls and data migration due to resource changes, the execution time, performance, and resource utilization efficiency in a static and elastic environment. With each application possessing certain representative characteristics, we generalize conclusions for Elastic MPI software development for specific applications types.

1.6 Outline

This dissertation is organized into four parts.

Part I provides introductory information. Chapter 2 reviews a few related research efforts that aim to solve similar problems.

Part II builds a foundation for understanding the Elastic MPI programming framework. Chapter 3 discusses the elastic MPI infrastructure including the Elastic MPI library programming API as well as the supporting implementation of the resource manager. This chapter provides necessary knowledge for all the following chapters. Chapter 4 summarizes generic programming models for different types of parallel applications. It also discusses how to handle applications with single and multiple resource-elastic phases.

Part III introduces three parallel applications developed in the Elastic MPI framework. Chapter 5 presents a malleable 2-D tsunami simulation that is transformed from a resource-static software. Chapter 6 presents a malleable 3-D oil reservoir simulation that

is also taken from the same resource-static software as the tsunami simulation. Despite the similarities, this application demonstrated very different behaviors with resource adaptivity. Chapter 7 discusses a statistical inverse problem solver with resource-elastic surrogate model construction.

Part IV consists of one chapter (Chapter 8), which summarizes our findings and gives an outlook for future Elastic MPI development.

2 Related Work

There have been several past and ongoing research efforts in elastic execution models. In this chapter, we cover some of the related work done in recent years. Since the present work is focusing on distributed-memory systems, we cover only research projects that support distributed-memory architectures, and leave out those aimed at shared-memory architectures as they are not so comparable with this work. Besides research projects, we also cover a very relevant but different technology – Cloud computing, which has gained vast popularity in recent years and is often compared or seen as an alternative to HPC.

Conventional HPC programming languages and models are resource-static: they operate with the configuration that resources never change during program execution, and therefore have no support for runtime resource changes. Any attempt to change this must provide some abstractions to represent resource changes. Reconfiguring resources and moving data over communication network must be taken care of very carefully, because these operations can introduce significant overhead that defeats the purpose of achieving high performance. In addition, support for resource elasticity must be added to applications as well as the system's resource management components. It is not uncommon that research on elastic execution models are coupled with scheduling research.

2.1 Dynamic Processes Support by the MPI Standard

Different implementations (both open-source and proprietary) of the MPI standard exist, such as MPICH [14], Open MPI [15], Intel MPI Library [16] and IBM Spectrum MPI [17]. Though their internal implementations differ, they all provide the same set of standard operations with the same interfaces. The standardization of MPI allows for source code level compatibility and portability across HPC systems.

The original MPI programming model is designed to be static in terms of number of processes, i.e., the number of processes that an MPI program starts with remains unchanged until the end of program execution. Since the MPI standard version 2.0, support for dynamic processes has been added, mainly through two additional operations, i.e., the `MPI_Comm_spawn` and the `MPI_Comm_spawn_multiple`. With these two and a few other auxiliary operations, additional processes can be added to MPI applications at runtime. However, there are limitations and shortcomings in the MPI standard:

- **Resource expansion is initiated by applications**, which have neither information on available resources nor the current state of other running applications. If the system is fully loaded, spawn operations could result in delay of execution, operation failure or creation of virtual processes depending on the implementation.
- **The spawn operations are blocking operations**. The parent processes[1] are blocked while the child processes[2] are being created. The delay in process creation can be significant and can degrade performance.
- **The spawn operations produce intercommunicators based on disjoint process groups (parent and child)**, each of which has its own `MPI_COMM_WORLD`, which means that the global communicator no longer reflects the global view of resources after spawn operations. Variables related to `MPI_COMM_WORLD` such as the communicator size and rank ID are no longer valid. This introduces complications to software development, and limits the re-usage of preexisting code.
- **Subsequent usage of spawn operations produce multiple intercommunicators and disjoint process groups**, which makes it hard to manage resources from a programming standpoint, especially for dynamic applications in which the number of spawn operations is runtime-dependent.
- **Processes created with spawn operations are typically run in the same resource allocation**, which limits the usefulness of these operations. This is not a deficiency of the MPI standard itself, but most implementations that support these spawn operations implement them in such a way that the spawned processes are created within the already allocated resources. In other words, there are no physical resources added but merely virtual processes that share the same resources.
- **Last but not least, runtime resource reduction is not supported**. Destruction of processes is only possible with `MPI_Finalize`, which means removing processes is not possible during runtime. Expansion-only resource elasticity is of little help for most of the current HPC problems.

Compared to the dynamic processes support from the MPI standard, Elastic MPI excels in design and implementation. Elastic MPI implements a centralized design, in which the resource manager, being aware of the state of all resources as well as the running and awaiting applications, initiates all resource adaptations. This eliminates the situations that no resources are available for resource expansion. In Elastic MPI, child processes are created asynchronously, while parent processes can continue to work without blocking. This implementation purposely hides latency to minimize resource adaptation overhead. Resource adaptation in Elastic MPI produces no additional communicators or process groups. It directly modifies `MPI_COMM_WORLD` to reflect resource changes, allowing for unlimited numbers of resource adaptations without complicating code development and maximizing re-usage of preexisting code. Successful expansion in Elastic MPI results in the inclusion of additional compute nodes (physical resources). And, last but not least, Elastic MPI supports resource reduction. The resource manager can freely instruct applications to expand and shrink to achieve optimal resource utilization efficiency.

[1] Processes in the parent process group, which is the group of original processes that call spawn operations.
[2] Processes in the child process group, which is the group being created due to spawn operations.

2.2 MPI Sessions

MPI Sessions [18] is a proposed interface extension for a future version MPI standard (MPI 4.0 or 4.1). It is currently under active development by a dedicated working group from the MPI Forum [4]. The key idea of this new interface is to introduce a concept of isolation in order to eliminate the requirement of a global communicator.

Holmes *et al.* pointed out that the management of a potentially massive process space is one of the major challenges that current MPI implementation and applications are facing [18]. The current requirement that all communication peers must be included in an immutable global communicator `MPI_COMM_WORLD` has become a scalability and performance barrier. With the observation that the common communication patterns in most MPI applications do not span all MPI processes, Holmes *et al.* proposed a fundamental change in MPI process management by introducing a layer of isolation for process spaces and relaxing the requirement for a global communicator.

`MPI_Sessions` are immutable local handles that contain no global states. They are intended to be light-weight and inexpensive to create or maintain. Each MPI Session forms an isolation domain, in which communicators and process groups are created. While an MPI Session itself is immutable, it can be created and destroyed at any time during the execution of a program. This introduces a lot of flexibility in MPI process management.

Even though resource elasticity is not the direct goal or motivation of MPI Sessions, its approach to addressing the MPI scalability issues – by introducing a flexible layer in MPI process management – overlaps with the goal of this work. With a proper elastic infrastructure support (on physical resources management), the MPI Sessions interface can be adopted for resource-awareness and elasticity realization. For instance, resource representation can be built upon an MPI Session. Whenever there is a change in physical resources, a new MPI Session (representing the new resource assignment) can be created and the old one can be destroyed.

With that said, there are fundamental differences between the MPI Sessions interface and the Elastic MPI framework. First of all, MPI Sessions is an interface only, it does not include or tie to any resource management infrastructure. Secondly, the interface of MPI Sessions eliminates the global communicator. Resource changes can be represented by the creation and destruction of MPI Sessions. On the other hand, the Elastic MPI framework preserves all current MPI interfaces including the global communicator. Resource changes are represented by a transmuted global communicator `MPI_COMM_WORLD`.

2.3 User-level Fault Tolerance in MPI

There have been many research work in fault-tolerant programming models for MPI applications. Some of the most recent ones include ULFM [19, 20], Fenix [21], FA-MPI [22, 23], Reinit [24], EReinit [25] and FMI [26]. Some of these methods have been proposed for the MPI standard, but none is yet adopted.

The User-level Failure Migration (ULFM) is a set of extension APIs to introduce fault-tolerance constructs to MPI. It provides interfaces for detecting process failures and repairing (rebuilding) communicators. The Fenix framework uses the ULFM interface to implement a global-restart model. It recovers damaged communicators via the PMPI interface such as the *shrink*, *spawn*, *merge* and *split* operations. It relies on user-registered

checkpoints for data recovery. Fault-aware MPI (FA-MPI) is also a set of extension APIs that provide fault detection and notification, assistance for isolation, and recovery procedures. Reinit also implements a global-restart model. After faults have been detected, it provides a mechanism to reinitialize MPI that returns a state similar to that returned by `MPI_Init`. EReinit uses the Reinit interface and takes advantage of the resource manager to implement failure detection, propagation, and recovery primitive. Fault Tolerant Messaging Interface (FMI) provides a messaging runtime that can survive process failures. It provides fast recovery via in-memory checkpoint-restart, scalable failure detection and dynamic spare node allocation.

All these fault-tolerant approaches provide mechanisms to facilitate the underlying resource changes at runtime, which is that some resources become unavailable during program execution due to hardware failure. In this regard, they overlap with the approach taken in this work to a certain extend. However, these fault-tolerant interfaces can only facilitate removal and replacement of resources. They lack support for resource expansion.

2.4 Charm++ and Adaptive MPI

Charm++ [27, 28, 29, 30] and Adaptive MPI [31, 32, 33] are parallel programming research projects aimed at providing dynamic load balancing support. Charm++ is a C++ based parallel programming system that implements the *message-driven* execution model with migratable objects. It provides a runtime system that can mitigate load imbalances caused by the dynamic behavior of applications. Adaptive MPI is the implementation of MPI on the Charm++ system.

In message-driven execution, processes are activated by messages. A process may utilize resources (processors) for execution only when it receives a message and that it can continue to work. If it blocks due to waiting for messages, other processes may utilize its processor for execution. If each processor is deposited with multiple processes, this strategy overlaps communication with computation seamlessly, as blocking communication no longer block resources.

In Adaptive MPI, ranks are implemented as lightweight user-level migratable threads instead of operating system processes, they are associated with Charm++ objects. The runtime system of Charm++ schedules multiple ranks to each processor, and moves them among processors or nodes when necessary to achieve good load balance. With the support for rank migration, fault tolerance can be achieved via a checkpoint-restart scheme.

Resource elasticity can also be achieved thanks to the rank migration capability. Kale *et al.* [34] presented a programming system for creating malleable jobs using Charm++ and Adaptive MPI. This scheme was based on a time-shared concept, in which performance of the system would improve with the number of jobs submitted. They demonstrated their concept with a molecular dynamics application which was embarrassingly parallel. Gupta *et al.* [35] proposed a scheme to realize runtime resource expansion and reduction via a combination of task migration, load balancing, Linux shared memory (SHM) and checkpoint-restart, under the assumption that there were always more tasks than processors. They demonstrated their approach with four mini-applications, three of which were 2-D or 3-D Cartesian grid based and one was embarrassingly parallel.

The Charm++ and Adaptive MPI projects strive to maximize system performance by providing dynamic execution support with a novel parallel programing system that in-

cludes a runtime scheduler and a programming Application Programming Interface (API). In this regard, it is very similar to our work. However, it is very different by design and by implementation. In Elastic MPI, ranks are normal operating system processes, just like in a normal MPI program. Indeed, Elastic MPI preserves everything from the the MPI standard, and achieve resource elasticity by an extension of four additional functions. Resource adaptation in Elastic MPI does not rely on SHM or checkpoint-restart. In Charm++, realization of dynamic load balancing replies on the over-decomposition of tasks to processors. This restriction does not exist in Elastic MPI. However, Elastic MPI does not provide automatic data migration and load balancing. Like in normal MPI programming, it is the programmer's responsibility to manage data movement and load balancing.

2.5 High Performance ParallelX (HPX)

The High Performance ParallelX (HPX) [36, 37, 38] is a general purpose task-based runtime system that strives for scalability on distributed-memory systems. It implements the ParallelX execution model [39, 40] and provides a programming framework that allows for transparent utilization of available resources. It is designed for conventional architectures with strict adherence to the C++11/14 standard.

HPX utilizes the governing principles of latency hiding, fine-grained parallelism, constraint based synchronization, adaptive locality control, work following data[3] and message driven[4], the combination of which allows for dynamic resource management, elastic execution, and in principle, efficient utilization of petascale and exascale machines.

HPX provides a single global address space across the system, enabling seamless load-balancing and dynamic adaptive resource management. The concept of *locality* is its means to express Non-Uniform Memory Access (NUMA): a locality is equivalent to a cluster node, intra-locality refers to local memory access, and inter-locality refers to remote memory access. First class objects such as threads and processes are decoupled from their locality and made migratable. Workloads are expressed as HPX-threads with immutable global names, which allow for remote task management. The HPX thread manager schedules HPX-threads onto a pool of OS threads (usually pinned to physical cores), removing overheads associated to OS thread creation. Blocking of a HPX-thread does not block the OS thread, allowing resources to be efficiently utilized by other workloads. The concept of a *parcel*, the remote semantic equivalent to a local HPX-thread, is used for inter-locality communication. Parcels can be used to move the work to the data or to gather data back to the caller.

Anderson *et al.* [41] presented an AMR based application with HPX implementations to eliminate global barriers. They showed that HPX is better capable of expressing finer-grained dependencies than MPI, at a cost of higher overhead and the use of more HPX-threads. The performance results, obtained on a 20-core machine, showed that the HPX version outscales and outperforms the MPI version when the two competing factors are properly balanced.

[3] Instead of moving data to the processor that executes the task (that requires the data), move the task to the processor that holds the data it needs. Moving tasks (a collection of program commands) instead of data is more efficient if the size of data is large.

[4] a computing style in which computation is invoked by the presence of messages, which convey both instructions and data

Compared to Elastic MPI, HPX employs a very different approach for resource efficiency. Elastic MPI takes the conventional practice of pinning parallel processes to resources. The realization of resource elasticity and efficiency is through dynamic resource allocation, i.e., adding and removing processes to applications at runtime. HPX decouples its workloads (threads) from resources. It schedules threads dynamically and moves them across resources. Threads do not block resources. Resource efficiency is achieved by managing threads dynamically.

2.6 Invasive X10 (*i*X10)

X10 [42, 43, 44] is an open-source object-oriented programming language that follows a Partitioned Global Address Space (PGAS) programming model. The development of X10 is an ongoing research project by IBM Research with the goal of providing a programming model that can address the modern architectural challenges. X10 shares many similarities with its peer PGAS languages, such as the concept of distributed arrays, constructs for parallel execution and synchronization, a two-layered memory model for local and remote memory, and more. Native X10 has potential for resource-elastic support, because it provides abstractions to represent resources in a NUMA domain.

X10 is one of the core programming languages supported by the *Invasive Computing* project. To achieve resource awareness and elasticity, the project extended X10 by adding function interfaces to support the action of resource inclusion and exclusion, as well as the interactions between elastic applications and runtime system. The invasive version of X10 is called *i*X10 [45, 46, 47]. Other constructs are also added to represent different resource types available in the hardware architecture developed by the *Invasive Computing* project. Dynamic resource management support is provided by the operating and runtime system developed by the project.

The major functions that enable the resource awareness and elasticity concept are `invade`, `infect` and `retrieve` `invade` allows applications to request more resources from the runtime system. They may or may not get the requested amount, depending on resource availability. If they do get additional resources, `infect` allows them to adapt execution to new resources. `retrieve` allows applications to release excess resources that are no longer needed. Applications can expand and reduce in resources multiple times by calling these functions as many times as necessary.

Though both *i*X10 and Elastic MPI are part of the *Invasive Computing* research project, Elastic MPI differs from *i*X10 in both design and implementation:

- *i*X10 implements a de-centralized design, in which applications initiate resource changes, and the runtime system reacts to their resource requests and makes final decisions. Elastic MPI implements a centralized design, in which the resource manager makes and casts resource decisions, and applications react to these decisions.
- *i*X10 and its paired operating and runtime systems are implemented for the special NUMA architecture developed by *Invasive Computing* . Elastic MPI is implemented for general distributed-memory HPC systems.
- *i*X10 implements a PGAS model while Elastic MPI implements the Message Passing programming model.

2.7 SALSA and PCM Extensions with IOS

Desell *et al.* [48] described a framework for application malleability, which enables applications to dynamically redistribute data as well as add and remove processing entities at runtime. For programming API, they implemented two language extensions, each of which was demonstrated with a scientific application. For dynamic resource management, they extended the Internet Operating System (IOS) [49] as a modular middleware. By comparing their implementations to a stop-restart scheme, Desell *et al.* showed that resource reconfiguration using their malleable framework were significantly faster.

The first malleable programming interface was an extension to SALSA (Simple Actor Language System and Architecture) [50, 51], which is a JAVA based actor-oriented programming language designed to facilitate the development of dynamic distributed applications. An astronomy application that uses linear regression techniques was made malleable with the extended SALSA. This application is embarrassingly parallel and it implements a master-worker model.

The second malleable programming interface was an extension to the PCM (Process Checkpointing and Migration) [52], which was a library previously developed by the same research group to enable process migration in MPI. In this interface, a master process (usually rank 0) is responsible for initiating data redistribution and updating all the necessary references. Resource adaptation occurs only at synchronization points. A grid-based application that models heat transfer in a solid was made malleable with the extended PCM. This application was of SPMD type with spatial data dependency.

The work by Desell *et al.* allowed for resource elasticity by providing programming interfaces and a supporting resource management infrastructure. In this regard, it is very similar to our work. And the PCM extension is a more comparable part to Elastic MPI, as it is also MPI based. The Difference is, in Elastic MPI, there is no need for a master process to initiate data redistribution. Data migration is orchestrated by all processes. Another difference lies in the evaluation focus. They evaluated their work by comparing the performance between their malleable framework and a stop-restart scheme. In this thesis, we are focused on the impacts of malleability on resource utilization efficiency on both the application- and system-level. We are more interested in comparing the malleable executions with the resources-static counterparts.

2.8 Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM) [53, 54, 55, 56] is a framework that connects a collection of Unix and Windows computers together by a network to be used as a single parallel machine. The project is designed to facilitate computation of large problems in a more cost efficient way by enabling users to exploit and aggregate their existing computer hardware.

The nodes (computers) in a PVM parallel machine is managed by users and can be modified at runtime, which provides support for resource elasticity and fault tolerance to some extent. PVM implements the Message Passing model and supports distributed-memory systems. Operations to spawn tasks, coordinate tasks and modify the parallel machine itself are provided. Applications can initiate resource expansion by requesting new tasks.

In a PVM parallel machine, a daemon is started before an application can be run on a node. The daemon acts as a local resource manager for the application. The application is linked to PVM's runtime library, which provides the implementation of the PVM API. Once the application is started, it can request to spawn more new tasks. Like in MPI, each task has a unique identifier and can exchange messages with other tasks.

The fact that PVM provides no coordination with resource managers makes it inadequate for resource-elastic systems with multiple users. However, resource-elastic behavior can be achieved on single jobs.

2.9 Cloud Computing

Cloud computing refers to utilizing computing services over the Internet ("the Cloud"). It has gained immense popularity in recent years due to its highly competitive cost-effectiveness. There exist both resource management and programming models in Cloud computing that allow for resource elasticity [57, 58, 59] and fault tolerance. Indeed, these technologies are mature and have been quite successful. On the contrary, development on resource-elastic support and fault tolerance in HPC are still in the early stages. The reason for this can be ascribed to the differences in application requirements on these systems.

Parallel applications run on Clouds have low requirements on communication and synchronization. Most of them are embarrassingly parallel to a certain extent. And many of them follow the client-server or MapReduce model. The client-server model [60, 61] refers to the parallel structure that partitions workloads between servers (service providers) and clients (service requesters). It is mostly used in web services. No dependency or synchronization requirements across user sessions makes it possible to employ this model. MapReduce [62, 63, 64] is a programming model that consists of a *map* operation, which performs tasks such as filtering or sorting, and a *reduce* operation, which performs summary or aggregation tasks such as counting the total number. Both map and reduce operations can be executed in parallel with no requirements for communication or synchronization. The model is mostly used in data analysis and processing algorithms.

Low requirements on communication and synchronization allow Cloud computing platforms to be designed more economically with commodity networks. Many Cloud service providers offer computation services in time sharing mode, in which resources are shared among multiple concurrent users. These designs cut down service costs by trading off predictability and reliability of the execution environment, which is tolerable by Cloud users due to the nature of their applications.

For many years, HPC systems have been prevalent platforms for large complex applications from science and engineering. These parallel applications have strong demand for communication and synchronization. Typically, computation domains in these applications are decomposed and distributed across processing units. Due to the nature of these scientific applications, data dependency usually exists and often requires periodic communication and synchronization between processing units. This is the major reason why HPC applications do not fit well in those programming models in Cloud computing.

To minimize the impact of communication and synchronization, HPC systems are designed with high performance networks that have lower latencies and higher bandwidths. Moreover, they typically operate in space sharing mode, in which applications have exclusive access to preallocated resources. These designs provide a very reliable and predictable

execution environment, however, they introduce higher purchase and maintenance costs compared to Cloud services.

In summary, even though resource elasticity is available in both resource management and programming models in Cloud computing, these technologies are not directly transferable to HPC due to the nature of most HPC applications. More research efforts are expected to advance technologies for resource-elastic execution and fault tolerance in the HPC world.

2.10 Summary

In this chapter, we discussed related research work that pursue similar goals or take similar approaches.

- The **MPI dynamic process support** allows for spawning more processes at runtime. However, it does not support removal of processes and it introduces complexity and difficulties in the programming experience. It is a pure interface that is not linked to any resource management infrastructure. Many MPI implementations create the spawned processes upon the already allocated resources, which means these is no actual change in physical resources.
- **MPI Sessions** is a proposed interface extension for the MPI standard to address the scalability issues by introducing an isolation layer for process management and relaxing the requirement for a global communicator. It has the potential to support resource elasticity if it is integrated with an elastic resource management infrastructure.
- Many MPI fault-tolerant programming models/interfaces, such as **ULFM**, **Fenix**, **FA-MPI**, **Reinit**, **EReinit** and **FMI**, provide mechanisms to facilitate runtime resource changes, i.e., certain resources become unavailable due to hardware failure and must be removed or replaced from the communication process group. Though allowing for resource removal and replacement, none of them supports resource expansion.
- The **Charm++/Adaptive MPI** project is a task-based parallel framework that implements a message-driven programming model. Its realization of resource efficiency relies on over-decomposition of tasks to processors, and resource elasticity is achieved with SHM and a checkpoint-restart scheme.
- **HPX** is also a task-based parallel system that implements a ParallelX programming model. It realizes resource elasticity and efficiency by implementing fine-grained tasks and moving them among resources automatically.
- ***i*X10** is a parallel programming framework that directly supports resource awareness and elasticity. It implements a PGAS programming model and is designed for a specific multiprocessor system-on-chip architecture.
- The **PCM extension with IOS** provides an elastic MPI programming model. Its resource adaptivity is initiated by a master process.

- **PVM** connects a set of computers to work as a parallel machine, in which resource expansion for an application is supported. However, it is disconnected from the resource manager and does not support multiple users.

Compared to all of these projects, Elastic MPI is unique as it implements a universal Message-Passing programming model by extending the MPI standard library. Like MPI, it provides interfaces for both C/C++ and Fortran languages, and is designed for general Linux-based architectures. It realizes resource elasticity and efficiency based on true dynamic resource (de)allocation and the support for runtime acquisition and removal of physical resources. It does not rely on checkpoint-restart or SHM. The system supports both elastic and static MPI tasks and multiple concurrent tasks (users).

Cloud computing, as a different technology, is often compared to HPC. Even though technologies for resource elasticity in Cloud computing have matured, they are not directly transferable to HPC as they are designed for applications that possess characteristics that most HPC applications do not. Elastic execution support tailored for HPC systems and applications are still in development.

Part II

Elastic MPI Framework

3 Elastic MPI Infrastructure

An infrastructure supporting resource awareness and elastic execution is one of the two key components of our solution to achieving resource efficiency in distributed-memory HPC systems. In this chapter, we cover the high-level design, functionality and implementation of the Elastic MPI infrastructure, which was first introduced in [12, 65]. This information lays a foundation for malleable parallel software development.

First, we take an overview of the infrastructure design. Then we discuss the Elastic MPI library and the elastic execution support in the resource manager, following with topics on pattern detection, performance monitoring and runtime scheduling. More implementation details of the Elastic MPI library (discussed in Section 3.2) and resource manager (discussed in Section 3.3) can be found in [12]. Lastly, we cover the assumptions and limitations in the current infrastructure implementation.

3.1 Overview

System-level optimization on throughput and energy efficiency is the ultimate goal of Elastic MPI. Compliant to this objective, the Elastic MPI infrastructure implements a *centralized design*, in which resources and applications are managed by a central component that oversees all workload and resource dynamics. This central component, in HPC systems, is called the *resource manager* or the *scheduler*, which is implemented as a separate software package from the operating system. In Elastic MPI, the resource manager takes care of launching and running parallel tasks, making decisions for resource assignments, and initiates all runtime resource changes.

The opposite of the centralized design is the *de-centralized design*, in which each parallel task optimizes for its own need by initiating resource change requests. In this scenario, the resource management unit merely reacts to the resource requests instead of making proactive decisions. While the de-centralized design offers opportunities for optimizing on the application level, it lacks the means to facilitate system-level optimization. In contrast, the centralized design pursues system-level optimality by trading off application-level optimality, which aligns with our objective.

Figure 3.1 presents an overview of the Elastic MPI system, its major components and their interactions between one another. In the center of the figure lies the resource manager, which interacts with resources and applications (batch jobs).

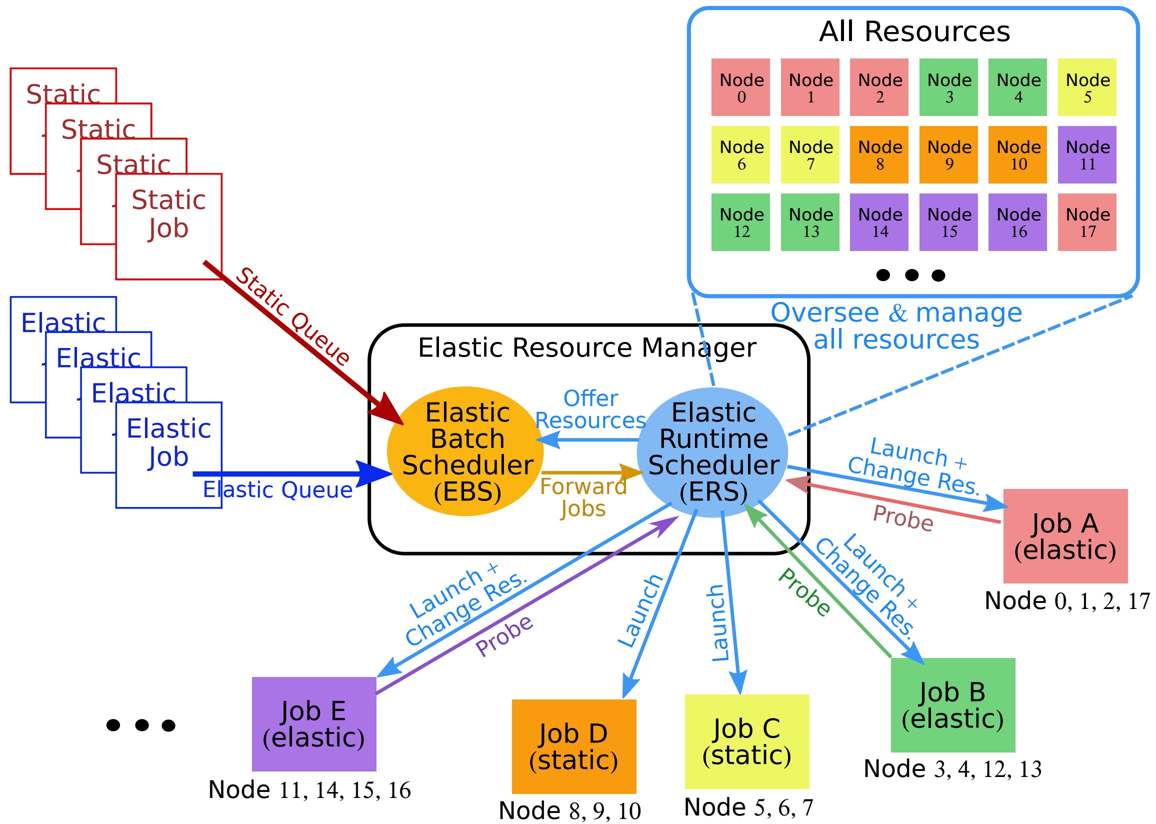


Figure 3.1: Overview of the Elastic MPI System: interactions between resource manager, resources and batch jobs. The resource manager consists of an elastic batch scheduler (EBS) and an elastic runtime scheduler (ERS). The EBS manages a static and an elastic job queue, and interacts with the ERS, i.e., if the ERS offers resources for launching new jobs, it decides which jobs to launch and forwards them to the ERS. The ERS oversees and manages all resources, and takes care of launching and running jobs. When resources become available, it makes decisions on either offering them for launching new jobs, assigning them to some of the running elastic jobs, or both. It also makes runtime decisions on resource reallocation according to the scalability of the running elastic jobs. It communicates its decisions with the EBS and all the running elastic jobs, which are expected to take actions accordingly. For static jobs, it only launches them and let them run to completion without interaction.

When a parallel task is submitted to a HPC system for computation, it first waits in a job queue for resource assignment, as it is required before execution. A dedicated component called the *batch scheduler* inside the resource manager handles the in-coming jobs. It manages two job queues: one for normal (static) jobs, and the other for elastic jobs. Its main functionality is to select the right jobs to launch. When there are resource offerings for new launches, it picks jobs from both queues based on their sizes (requested resources), priority, fairness (e.g. how long has the job been waiting) and other policies enforced by the system administrator. The goal is to fully utilize the resource offerings under policy constraints.

The selected jobs are then forwarded to another component of the resource manager – the *runtime scheduler*, which oversees and manages all resources in the system. It allocates resources and launches the forwarded jobs. For the static ones, it simply lets them run to completion without interaction. For the elastic ones, it periodically monitors their parallel performance and makes runtime resource change decisions accordingly. When more resources become available, it decides between offering them for launching new jobs and/or assigning them to currently running elastic jobs that have good scalability.

The periodic resource allocation decision is made for all running elastic jobs at once. For some jobs, their resource allocations stay the same, therefore, no actions are required from them. For others, there are resource changes, and thereby, taking immediate actions to adapt is required. These decisions from the runtime scheduler are communicated via probing actions by the jobs. Elastic jobs are expected to communicate with (probe) the resource manager frequently and take prompt actions accordingly. These communication and resource adapt actions are facilitated by the Elastic MPI library.

3.2 Elastic MPI Library

Designed with the goals of latency hiding, minimal collective latency and ease of programming, the Elastic MPI library is implemented upon an MPICH v3.2 base [14] with a set of four new operations:

- `MPI_Init_adapt`: for initializing MPI processes in adaptive mode,
- `MPI_Probe_adapt`: for probing for resource change decisions,
- `MPI_Comm_adapt_begin`: to indicate the beginning of resource adaptation,
- `MPI_Comm_adapt_commit`: to indicate end of adaptation and commit resource changes,

as well as the supporting code for their integration with the resource manager.

The new operations are exposed to programmers, as they serve to facilitate resource adaptation as well as the communication between applications and the resource manager. They overcome the limitations of the dynamic processes support by the MPI standard (discussed in Section 2.1), and realize efficient physical resource adaptation. Their supporting code for integration with the resource manager, on the other hand, is purposely hidden from programmers and excluded from the API.

The following subsections explain the interface of each of these new operations, as they are the premises for understanding the Elastic MPI programming models discussed in Chapter 4. Explanations and demonstration on how to use these operations to assemble resource-elastic applications can be found in Sections 4.2 through 4.5.

3.2.1 MPI Initialization in Adaptive Mode

The `MPI_Init_adapt` operation allows applications to initialize MPI processes in adaptive mode. This is how the resource manager distinguishes elastic applications from the static ones, i.e., only those initialized with `MPI_Init_adapt` are considered elastic, and would be monitored and interacted with for resource changes during execution.

```c
int MPI_Init_adapt(
    int *argc, char ***argv, int *proc_status);
```

Listing 3.1: `MPI_INIT_ADAPT` C interface

```fortran
SUBROUTINE MPI_INIT_ADAPT(proc_status, ierror)
    INTEGER proc_status
    INTEGER ierror
END SUBROUTINE MPI_INIT_ADAPT
```

Listing 3.2: `MPI_INIT_ADAPT` Fortran interface

Listings 3.1 and 3.2 show the operation's C and Fortran interfaces respectively. It has a similar interface as the `MPI_Init` operation, except for one additional output parameter `proc_status`, which returns the status of the process.

In Elastic MPI, we define four states (status) of processes:

- `MPI_ADAPT_STATUS_NEW`, which indicates that the process is created during the launcher command provided by the resource manager, e.g., `mpiexec`, `srun`, etc;
- `MPI_ADAPT_STATUS_JOINING`, which indicates that the process is created as part of a resource expansion;
- `MPI_ADAPT_STATUS_STAYING`, which indicates that the process will remain after a resource adaptation (expansion or reduction);
- `MPI_ADAPT_STATUS_LEAVING`, which indicates that the process will retire after a resource reduction.

It is crucial to distinguish between different status of processes, because different actions are required from each type to orchestrate a successful data migration and process integration in case of resource expansion. More details on this subject is discussed in Section 4.2 through Section 4.5, where generic Elastic MPI programming patterns are abstracted for different types of applications.

The output parameter `proc_status` in `MPI_Init_adapt` can take two possible values: either `MPI_ADAPT_STATUS_NEW` or `MPI_ADAPT_STATUS_JOINING`.

3.2.2 Probing for Resource Change Decision

The `MPI_Probe_adapt` operation allows the preexisting processes to probe the resource manager for resource change decisions. Runtime resource allocations are determined periodically by the resource manager for all running elastic applications. An elastic application might or might not need to adapt depending on whether its new resource allocation is the

same as the current one. In order to get such information, it is expected to probe the resource manager periodically and react to the decision accordingly.

```
int MPI_Probe_adapt(
    int *adapt_flag, int *proc_status, MPI_Info *info);
```

Listing 3.3: `MPI_PROBE_ADAPT` C interface

```
SUBROUTINE MPI_PROBE_ADAPT(adapt_flag, proc_status, info, ierror)
    INTEGER adapt_flag
    INTEGER proc_status
    INTEGER info
    INTEGER ierror
END SUBROUTINE MPI_PROBE_ADAPT
```

Listing 3.4: `MPI_PROBE_ADAPT` Fortran interface

The operation's C and Fortran interfaces are shown in Listings 3.3 and 3.4 respectively. The output parameter `adapt_flag` tells the application whether or not it should adapt. It gives the value `MPI_ADAPT_FALSE` for no adaptation, or `MPI_ADAPT_TRUE` to signify a pending adaptation.

The output parameter `proc_status` returns the status of the process, which is explained in Section 3.2.1. In case of resource adaptation, different actions are required from different types of processes in order to successfully perform data movement as well as process integration or exclusion. This parameter takes three possible values: `MPI_ADAPT_STATUS_JOINING`, `MPI_ADAPT_STATUS_STAYING`, or `MPI_ADAPT_STATUS_LEAVING`.

The output parameter `info` is optional and can be set to `NULL`. It returns an `MPI_Info` object that provides additional information from the resource manager.

3.2.3 Beginning Resource Adaptation

When the `MPI_Probe_adapt` operation returns a positive signal for resource adaptation, the application is expected to react promptly. To ensure a safe adaptation environment, in which the application has stable access to all resources, i.e., the `STAYING`, `LEAVING` and `JOINING` processes, the concept of *adaptation window* is introduced.

The adaptation window refers to the period between operations `MPI_Comm_adapt_begin` and `MPI_Comm_adapt_commit`, during which all actions related to resource adaptation should be performed. By calling `MPI_Comm_adapt_begin`, the application informs the resource manager that it begins the adaptation process and that all resources (`STAYING`, `LEAVING` and `JOINING` processes) should be ensured accessible until the process is completed. Within the adaptation window, the preexisting (`STAYING` and `LEAVING`) processes and the world communicator `MPI_COMM_WORLD` are preserved; the `JOINING` processes (if there are any) come as a separate group with their own world communicator.

```
int MPI_Comm_adapt_begin(
    MPI_Comm *intercomm, MPI_Comm *new_comm_world,
    int *staying_count, int *leaving_count, int *joining_count);
```

Listing 3.5: `MPI_COMM_ADAPT_BEGIN` C interface

```
SUBROUTINE MPI_COMM_ADAPT_BEGIN(inter_comm, new_comm_world,&
        staying_count, leaving_count, joining_count, ierror)
    INTEGER intercomm
    INTEGER new_comm_world
    INTEGER staying_count
    INTEGER leaving_count
    INTEGER joining_count
    INTEGER ierror
END SUBROUTINE MPI_COMM_ADAPT_BEGIN
```

Listing 3.6: `MPI_COMM_ADAPT_BEGIN` Fortran interface

The C and Fortran interfaces of the operation are shown in Listings 3.5 and 3.6 respectively. The first two output parameters provide two temporary communicators that can be used for data migration and synchronization. The `intercomm` returns an intercommunicator that includes both the preexisting and the joining process groups. The `new_comm_world` returns a communicator that represents the new resource allocation, i.e., the union of the `STAYING` and `JOINING` processes (if any) excluding the `LEAVING` processes (if any). Both of these communicators are intended to provide convenience for data movement during adaptation. Figure 3.2 shows the communicators before, during and after resource adaptation. The last three output parameters return the counts of the `STAYING`, `LEAVING` and `JOINING` processes respectively, from which the application can determine whether it should perform resource expansion or reduction.

3.2.4 Committing Resource Adaptation

The operation `MPI_Comm_adapt_commit` signifies the ending of the adaptation window and commits the resource changes. It does not have any return parameter. Listings 3.7 and 3.8 show the C and Fortran interfaces of the operation respectively.

Upon return, the world communicator `MPI_COMM_WORLD` would be updated with the temporary communicator `new_comm_world`, as shown in Figure 3.2. The `LEAVING` processes (if any) would be released and made available to other applications. The temporary communicators `intercomm` and `new_comm_world` from the `MPI_Comm_adapt_begin` operation would be deleted. The programmer must update the rank and size of the world communicator, since it has been changed. In addition, the programmer should also manually update the process status, i.e., assign `MPI_ADAPT_STATUS_STAYING` to `proc_status` for each process, so that they can be clearly distinguished from the `JOINING` and `LEAVING` processes in the next adaptation.

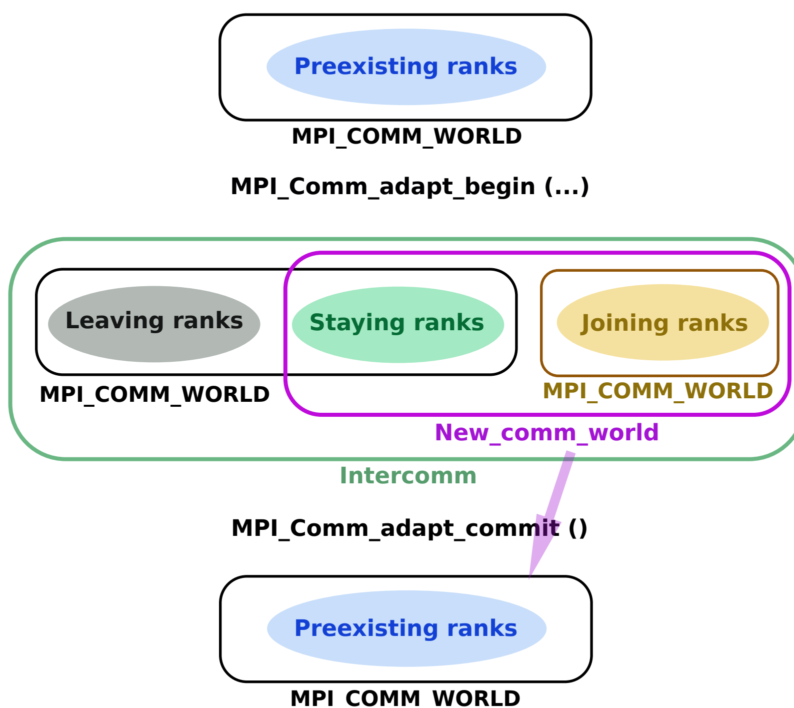


Figure 3.2: Elastic MPI communicators before, during and after resource adaptation

```
int MPI_Comm_adapt_commit();
```

Listing 3.7: `MPI_COMM_ADAPT_COMMIT` C interface

```
SUBROUTINE MPI_COMM_ADAPT_COMMIT(ierror)
    INTEGER ierror
END SUBROUTINE MPI_COMM_ADAPT_COMMIT
```

Listing 3.8: `MPI_COMM_ADAPT_COMMIT` Fortran interface

3.3 Elastic Resource Manager

HPC systems are designed to deliver high performance, therefore, they are expected to provide reliable and predictable execution environments. For this reason, most HPC systems operate in *space-sharing* mode, in which parallel jobs are guaranteed exclusive access to requested resources for their entire execution.

The counterpart of space-sharing mode is *time-sharing*, in which computational resources are shared among multiple concurrent users, tasks or programs. A common example of time-sharing is an everyday computer, which runs background system tasks, renders graphics, performs user tasks such as web browsing and gaming, all at the same time.

Under space-sharing mode, jobs must be first assigned resources before they can be executed. When a job is submitted to an HPC system, it is not started immediately. Instead, it is queued and executed at some time in the future when the resources it needs

become available. The starting time is not guaranteed as it is dependent on the workload on the system and the availability of resources. Once the job is started, it would run to completion without being interfered with.

In order to deliver performance that is commensurate to its hardware resources, an HPC system must optimize on these performance metrics, which is often a task delegated to the resource management component. The terms *resource manager* and *scheduler* is often used interchangeably. A resource manager takes a set of jobs to be executed and a set of available resources as input, and produces a performance optimizing order for launching jobs as output. These orders are referred to as schedules (hence the name scheduler), which affect the performance of the HPC system.

3.3.1 Requirements for Elastic Resource Management

Scheduling of job launches is the only decision logic in resource management required in current HPC systems. For runtime resource-elastic execution support, however, additional functions must be added to the resource manager:

- Besides batch scheduling (order to launch jobs), runtime scheduling must be provided. System-wide performance optimizing decisions on which applications are to expand and which ones are to shrink, as well as on when more resources become available, should they be added to running jobs or used to launch new ones, must be determined based on the scalability of the running elastic jobs.
- Elastic jobs need continuous performance monitoring. As most of them are expected to have dynamic workload behaviors, their parallel performance and scalability would change over time. Runtime resource decisions must be made based on their most updated performance measurements.
- Current HPC infrastructures operate under the configuration that a job's resource allocation is constant. To support runtime elastic execution, the infrastructure must allow modifications to resource allocations and reconfiguration of the running systems.
- Last but not least, communication between the resource manager and elastic applications must be supported. This is needed for the propagation of runtime resource decisions to elastic applications as well as coordination between the two parties to achieve successful resource adaptation.

3.3.2 SLURM: Base of the Elastic Resource Manager

The Simple Linux Utility for Resource Management (SLURM) [13] is an open-source resource management and job scheduling system for Linux and Unix-like clusters. It is a highly scalable solution and currently quite popular in HPC systems, especially among the world's TOP500 supercomputers [13]. It provides three key functions: allocate resources to jobs for some duration of time; start, execute and monitor parallel jobs on the set of allocated resources; manage a pending job queue and arbitrate contention for resources. After careful evaluation on popular HPC schedulers, SLURM was chosen as a base for the Elastic MPI infrastructure implementation.

SLURM can be seen as a collection of binaries that shares a single configuration file. It consists of the following major components: the controller daemon (`SLURMCTLD`), node daemons (`SLURMD`), application launchers (`SRUN`), and job step daemons (`SLURMSTEPD`). It also provides plug-in interfaces for extended functions such as scheduling, topology, MPI and Process Management Interface (PMI) support, database support, and more. Figure 3.3 shows an abstraction of SLURM internal organization of components, as well as the links between PMI and MPI libraries and MPI processes.

A *daemon* is a background process that acts as an agent to performance dedicated tasks. The controller daemon `SLURMCTLD`, including the batch scheduler as a modular plug-in, manages the security, user accounts, tracks individual nodes, and so forth. It is the only centralized component of SLURM, and is generally run on a dedicated node. Other user controls and binaries such as `SRUN`, `SBATCH`, `SCANCEL`, among others, interact with it over the network.

There is a node daemon `SLURMD` running on each node. Node daemons are used for controlling and monitoring individual nodes. They are also responsible for starting the job step daemons.

A *job step* is an application started with `SRUN` on a set of allocated resources. The resources of a job step are managed through an instance of the application launcher `SRUN` and a set of job step daemons `SLURMSTEPD`. Per job step, there is an instance of `SRUN` running on the master node (node that contains MPI rank 0) and a `SLURMSTEPD` running on each node.

3.3.3 Support for the Elastic MPI Library

SLURM manages resources by orchestrating the interactions and communication between the centralized daemon and the node-local and job-local components. It is implemented under the assumption that resource allocations are constant. To provide support for the extended elastic operations presented in Section 3.2 as well as the required functionality described in Section 3.3.1, modifications and extensions are made to various parts of the original SLURM.

Figure 3.4 provides an overview of the SLURM-based elastic resource manager, which also consists of centralized and local components. Interactions between components are demonstrated with a resource expansion case. The original controller daemon `SLURMCTLD` is replaced with two components: the *Elastic Batch Scheduler* (EBS) and the *Elastic Runtime Scheduler* (ERS). The EBS manages the batch scheduling and security that was originally provided by the controller daemon. The ERS provides various dynamic runtime supports that do not exist previously, such as runtime scheduling, performance monitoring, resource allocation modification, runtime communication, and many more.

For the `MPI_Init_adapt` operation, the application launcher `SRUN` receives the initial resource allocation and credentials from the ERS. It then creates MPI processes by interacting with the node daemons `SLURMD` of the allocated nodes. The information of process status `proc_status` are propagated to each process with its initialization metadata.

With the `MPI_Probe_adapt` operation, resource change decisions are forwarded from the ERS to the preexisting processes. This operation is coordinated by the `SRUN`. In case of resource expansion, preexisting processes continue with work while the expansion processes are being created. The preexisting processes are notified only when the expansion

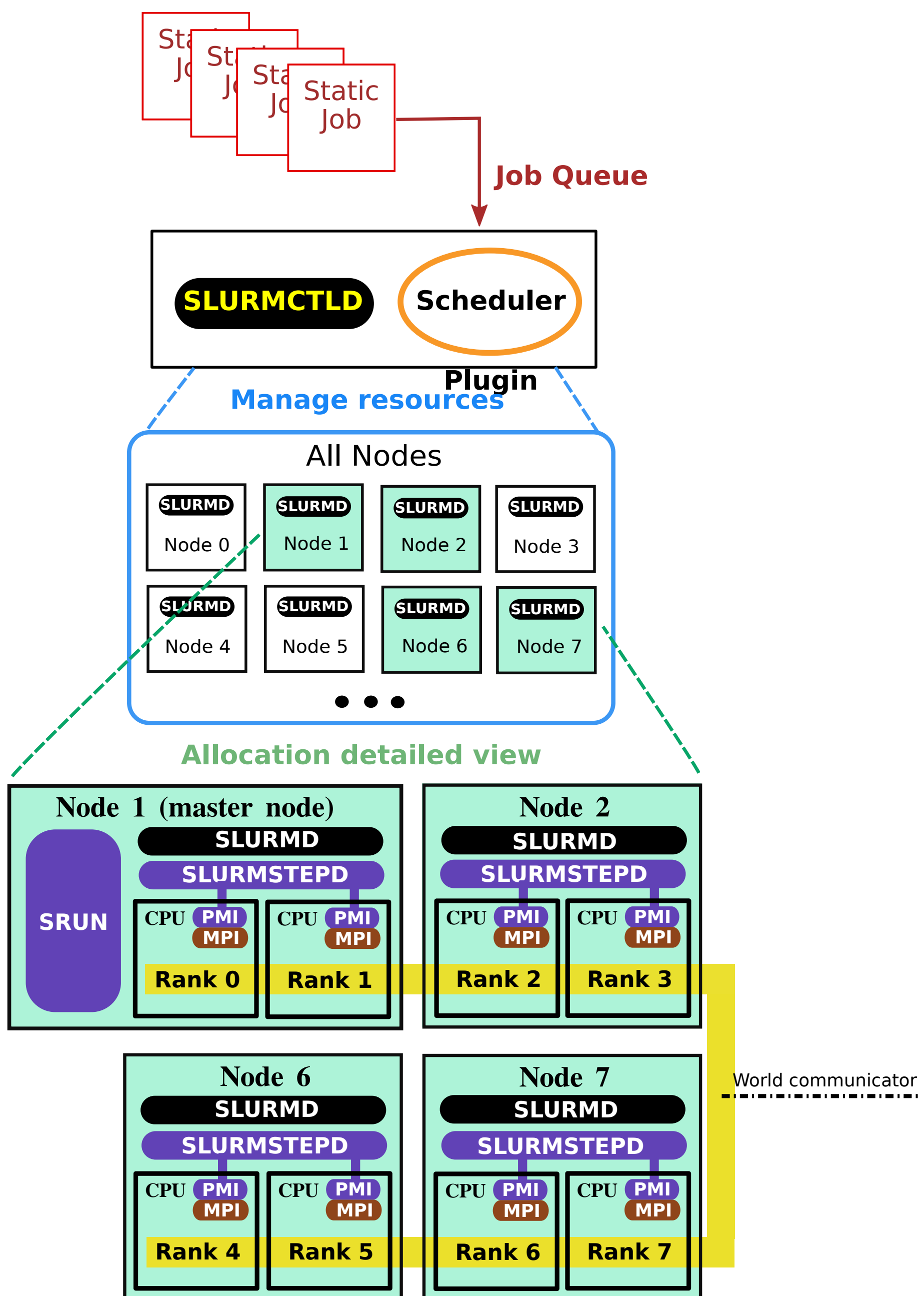


Figure 3.3: SLURM overview: abstract organization of SLURM components (SLURMCTLD, SLURMD, SRUN and SLURMSTEPD) on a cluster.

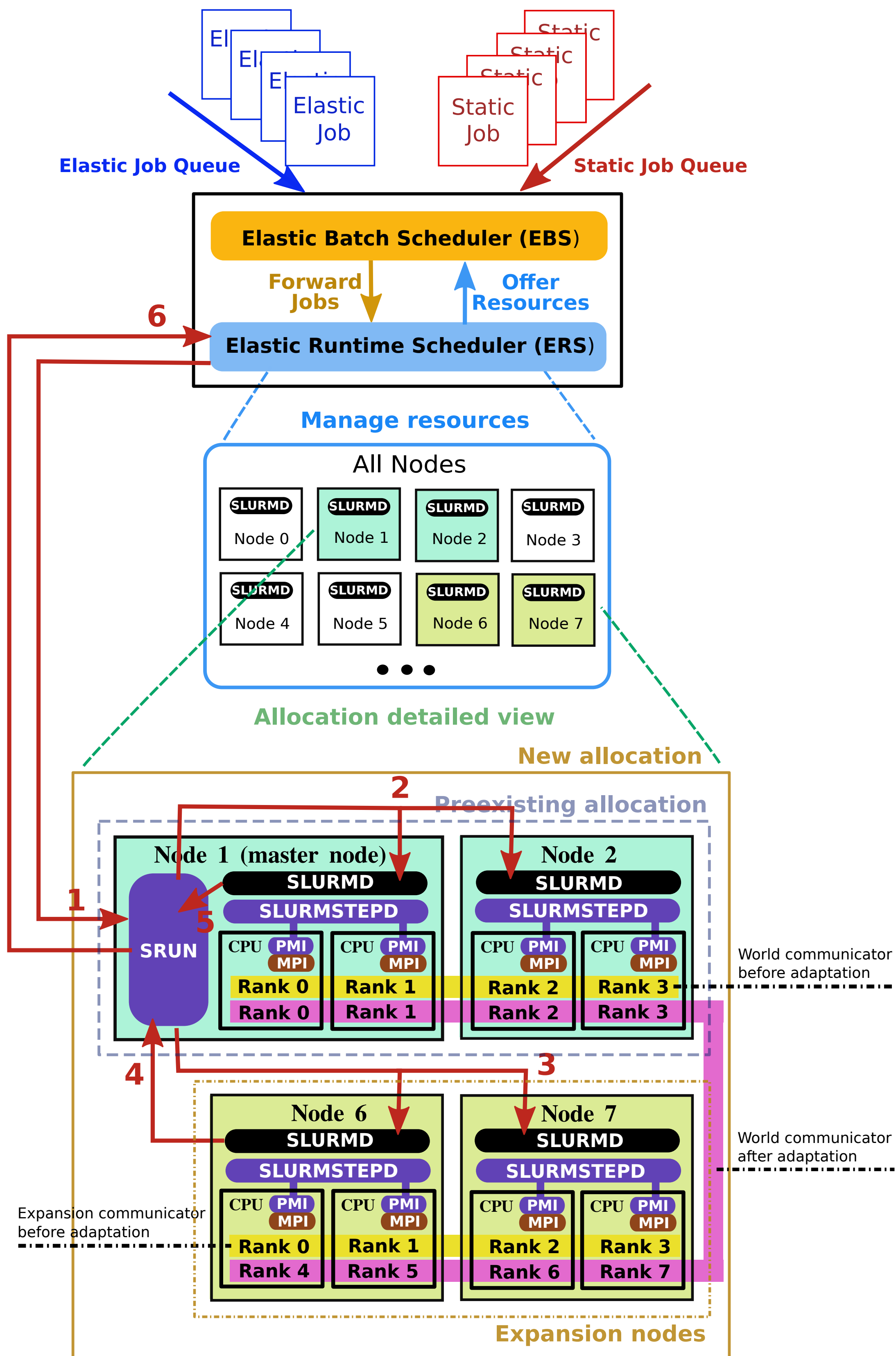


Figure 3.4: SLURM-based Elastic MPI infrastructure overview: internal component organization and interactions demonstrated with a case of resource expansion.

processes are ready. This latency hiding design minimizes the overhead induced by resource adaptation.

For completion of an adaptation window, the following six steps take place in the elastic resource manager:

1. The ERS makes a decision and sends a reallocation message to the `SRUN` instance of the job step.
2. The `SRUN` then sends instructions the `SLURMD` of the preexisting nodes. In case of resource reduction, these instructions tell some of the processes to leave the process group. Via operation `MPI_Probe_adapt`, `SLURMD` updates the metadata of each local MPI processes,
3. In case of expansion, `SRUN` sends a launch command with other required instructions to the `SLURMD` of each participating expansion node.
4. After the MPI processes in expansion nodes are created, `SLURMD` of the leading expansion node notifies `SRUN` in the `MPI_Comm_adapt_begin` operation.
5. Once the adaptation window is completed, upon calling `MPI_Comm_adapt_commit`, the `SLURMD` of the master node notifies `SRUN` that adaptation is completed.
6. `SRUN` notifies the ERS that resource changes are applied to the job step, and receives an updated credential with added nodes or leaving nodes removed from the allocation.

Each of the fore mentioned steps is marked in Figure 3.4 with an arrow pointing from the component that performs the action to the component that receives the action. Steps 3 and 4 only take place in case of resource expansion. Notice that there are two ranks associated with each MPI process: the one in the yellow communicator denotes the rank before adaptation commit, and the one in the purple communicator denotes the rank after.

3.3.4 Pattern Detection, Performance Monitoring and Runtime Scheduling

Optimizing system performance metrics relies on accurate performance prediction of jobs. For elastic jobs, this requires continuous real-time performance monitoring, which is a task delegated to the ERS in the Elastic MPI infrastructure. The ERS detects patterns of applications such as their parallel execution models and loop structures, collects their performance data, and periodically makes resource (re)allocation decisions based on these data.

Pattern Detection

In Elastic MPI, pattern detection [66, 67, 68] is achieved by the introduction of *markers* by the compilation wrappers, i.e., `mpicc`, `mpicxx` and `mpifort`, which currently work with GNU and Intel compilers. Using these markers as identifiers, structures of computation workflow, such as loops, can be identified. Minimization of performance impact is of great importance because pattern detection is intended to occur real-time in production runs of elastic applications. The injection of markers at compile time eliminates the overhead related to back-tracing.

The wrapper-based technique relies on the generated function calls in the emitted assembly. Once an MPI or PMI call is identified, a unique integer ID is computed and inserted

before the function call. This is done by an internal (not exposed to users) operation provided by the Elastic MPI library. Thanks to the fact that MPI and PMI operations can be easily identified, it is guaranteed that only these operations are intercepted.

Using these unique IDs (markers), structures of computation can be extracted at runtime. There are several algorithms for detecting patterns in sequences [69, 70, 71, 72, 73]. The one adopted in Elastic MPI is based on [74], which was originally designed to analyze programs from decompilation. The algorithm performs the following:

1. Generate a Control Flow Graph (CFG) based on detection.
2. Annotate each node of the CFG with its number of revisits.
3. Mark the head of each unique loop.
4. Mark the tail of each unique loop.
5. Mark the reentry points from each nested loop.

The CFG is represented in a text-based tabular form. MPI functions are categorized into different types such as point-to-point, one-sided, collectives, MPI-IO, among others. With its type and unique ID, a relevant MPI function call triggers update to the CFG. The detection logic is only available to elastic applications, i.e., the ones initialized with `MPI_Init_adapt`. Pattern detection is by default disabled, and is enabled after the first function call to `MPI_Probe_adapt`.

Performance Monitoring

Performance monitoring works side-by-side with pattern detection. MPI processes of an elastic application start to record performance data locally once any loop is detected with its head and tail identified. In the current implementation, two performance metrics are recorded: the *Total MPI Time* (TMT) and the *Total Loop Time* (TLT). The TMT is the total time spent on MPI operations. It is computed by accumulating the difference between the exit and entry time of each MPI function call. The TLT is the total time spent on the detected loop, which is inclusive of its TMT. It is computed by subtracting the loop creation time from the time of its last visit. The ratio of TMT to TLT of the critical loops gives a very good estimate of the real-time scalability of the application, on a process-level.

Each process serializes its local CFG data and sends it to the local node daemon `SLURMD`. `SLURMD` keeps track of the CFG data and their updates from all node-local processes. It performs CFG reduction whenever CFG data or an update is received. The TMT and TLT metrics of each process in a loop are added to the reduced loop head nodes.

The ERS periodically generates a performance data request that reaches all nodes of an elastic application. Indeed, the ERS requests performance data from all currently running elastic jobs at the same time. These requests and responses are routed by the application launcher `SRUN` of each elastic application. The ERS then performs a final reduction on the received node-local performance data and generates a performance model for each elastic application. The set of performance models are then used by the ERS to make resource reallocation decisions.

Runtime Scheduling

On each resource evaluation, the ERS performs the following operations. It first iterates the running job list and selects the elastic ones with available performance data. It generates a

performance model for each selected job, and computes a range of mandatory and optional resource adaptation. It makes a resource offer to the EBS (for launching new jobs) based on the computed range. It then performs *Elastic Backfilling* [12], an algorithm designed to reduce the number of idle nodes. Via the `SRUN`s, it applies its resource change decisions to every elastic job. Then it launches the new jobs forwarded by the EBS, if there is any. Finally, it waits until the system reaches a steady state before moving on to the next evaluation.

The computation of the resource adaptation range is a heuristic based on empirical data from initial experimentation. In this algorithm, a *Resource Range Vector* (RRV) is generated based on a metric called *MPI to Compute Time* (MTCT) of the current loop of an application. A high MTCT ratio indicates large communication time, thus hints for resource reduction. On the contrary, a low MTCT ratio hints for resource expansion. An average and a trend value of the MTCT are computed.

The RRV is generated based on the following rules:

- If any of the MTCT values is above the upper threshold, then reduce the application's resources to half.
- If one value falls in between the upper and lower thresholds and one is blow the lower threshold, then the application should stay unchanged and is thus removed from the set of candidates for changes.
- If both values are below the lower threshold, then double the application's resources.

The upper and lower thresholds can be changed in the Elastic MPI configuration. They create a bandwidth that helps prevent resource adaptation oscillations. The doubling and halving of resources can be replaced by more precise values if more sophisticated performance models are supported in the future. This algorithm tries to keep both MCTC values blow the upper threshold for all elastic applications, which means it reduces more aggressively. This is due to a design decision that we want to prioritize the launch of new applications over the expansion of running applications.

3.4 Limitations and Known Issues

3.4.1 Node-level Resource Granularity

The current Elastic MPI implementation supports node-level resource allocation and adaptation. Regardless of how MPI processes are mapped to physical resources, a node is either completely included to or excluded from a resource allocation. It cannot be utilized partially, nor be shared among multiple applications.

It is possible to modify the resource metadata representation stored on the node daemons to support finer granularity, e.g., core-level resource allocation. However, the impact of finer resource granularity on overall system performance requires further investigation. There is currently no plan to add such feature to a future Elastic MPI release.

3.4.2 Constant Master Node

The current Elastic MPI infrastructure supports arbitrary resource expansion and reduction, with one exception where the master node (on which `SRUN` is located) of an application

must remain in the allocation. This is due to the fact that SRUN cannot be migrated given the inherent design from SLURM. Adding a migration feature to SRUN would enable arbitrary migration of full elastic applications.

All elastic applications presented in this dissertation are implemented under the assumption that the master process (Rank 0) of an application remains unchanged throughout its execution. This is a safe and valid assumption in the current release, given that the master node, which contains Rank 0, remains in the allocation.

3.4.3 Rank to Process Mapping Strategy

In order to minimize data migration in resource adaptation, the current rank to process mapping algorithm is designed to minimize rank changes, i.e., the ranks of the STAYING processes are preserved. This is achieved by selecting the largest ranks for removal in case of resource reduction, and assigning the JOINING processes with ranks greater than the highest STAYING rank in case of resource expansion. Furthermore, to simplify the design and implementation, the current Elastic MPI version does not support the case of mixing reduction and expansion in one adaptation. An adaptation is either a pure expansion or a pure reduction.

All presented elastic applications in this work are implemented with such assumptions. Let N be the total number of current (preexisting) processes, N_j the number of JOINING processes, N_s the number of STAYING processes and N_l the number of LEAVING processes. In case of a reduction, $N_j = 0$ and $N = N_s + N_l$; it is assumed that Rank 0 to Rank $N_s - 1$ are staying, and Rank N_s to Rank $N - 1$ are leaving. In case of an expansion, $N_l = 0$ and $N = N_s$; it is assumed that Rank 0 to Rank $N - 1$ are unchanged, and Rank N to Rank $N + N_j - 1$ are the new joining ranks.

While this strategy guarantees minimal changes in STAYING ranks, it does not allow arbitrary removal of processes, which would be a requirement for fault-tolerant support. Should fault-tolerant support be implemented in a future Elastic MPI release, these assumptions would not hold true anymore.

3.4.4 Incomplete Functionalities of the Elastic Resource Manager

While the elastic resource execution support, i.e., the integration of the Elastic MPI library and the resource manager, is fully implemented, the following functionalities of the resource manager are not yet complete:

- The batch scheduler EBS is not provided in the current release.
- Only the SPMD execution model is implemented for pattern detection and performance monitoring in the runtime scheduler ERS. Other models (such as master-worker) are not yet supported. Application implementing a model other than SPMD would be treated as a static MPI application.

For these reasons, in this dissertation we could not include integration tests, in which multiple elastic applications run concurrently in the same environment to compete for resources. In the presented tests of individual applications, we used different pre-defined scheduling logic instead of the runtime scheduling based on pattern and performance data. This is partially due the specific needs of different tests, partially due to the incomplete support for different execution models, and partially due to some existing implementation defects in the infrastructure.

3.4.5 Stability Issues and Limited Scalability

Besides the MPI library and resource manager, other components are needed in the Elastic MPI infrastructure for the communication and interaction between different hardware and software layers. This increases the development complexity and difficulty.

There have been known stability issues that are unresolved or only partially resolved in different components, especially problems in the fan-out functionality in the PMI layer and problems with the internal metadata organization. Most of these issues are rooted in the original design for static resources in the SLURM and MPI bases, which makes it difficult to find radical solutions unless replacing the old design entirely. These issues majorly affect and limit the scalability of the infrastructure.

So far, tests on scalability (not presented in this work) have been successfully conducted on up to 128 and 64 nodes on two CPU clusters in the SuperMUC Petascale System [75], and those beyond that scale have failed. This is why tests presented in this dissertation were conducted on limited nodes. Resolving the scalability issues is the utmost important task in future Elastic MPI releases.

3.5 Summary

In this chapter, we discussed the high-level design and implementation of the Elastic MPI infrastructure, which consists of two major components: the Elastic MPI library and the elastic resource manager. The extended API provides four additional functions to facilitate resource adaptation operations as well as communications between elastic applications and the resource manager. The resource manager is implemented based on SLURM with modifications and extensions to provide required functionality for runtime resource changes. Limitations and known issues of the current Elastic MPI release were also elaborated.

4 Parallel Programming with Elastic MPI

In this chapter, we discuss resource-elastic programming abstractions. In particular, we investigate the two types of resource changes currently supported in Elastic MPI: pure expansion and pure reduction. A pure expansion consists of `STAYING` and `JOINING` processes, while a pure reduction consists of `STAYING` and `LEAVING` processes. In the current implementation of Elastic MPI, a mixture of `STAYING`, `JOINING` and `LEAVING` processes in one adaptation is not supported.

While the treatment for reduction is straight forward, i.e., saving the data from the `LEAVING` processes before they are released, more efforts are required to handle expansion as the `JOINING` processes start executing from the very beginning of the program with an empty function stack. The key is to integrate them with the preexisting processes with the least amount of work and time, which requires not only proper data preparation but also a correct routing strategy to bypass unnecessary operations.

4.1 Classification of HPC Applications

One of the fundamental steps in parallel computing is to decompose the problem into a set of tasks to be executed concurrently. Many decomposition techniques exist. As a wide classification, they can be domain decomposition or functional decomposition [76], which are also known as data parallelism and model parallelism respectively.

In a domain decomposition approach, the data (or computation domain) associated with the problem is divided into smaller subdomains, which can be processed concurrently. In contrast, in a functional decomposition approach, the computational tasks are decomposed into smaller subtasks, each of which can operate on the same set or different sets of data. In this work, we focus on data-parallel applications.

On distributed-memory systems, each parallel process has its own private memory space and is isolated from other processes in terms of data. It requires communication between parallel processes to exchange information. Such inter-process communication plays a key role in parallel performance due to the additional overhead introduced by sending and receiving data over the communication network.

Some problems can be decomposed in such a way that the parallel subtasks require little to no communication between one another. These types of problem are called *embarrassingly parallel*. On the contrary, computation of parallel subtasks in some problems requires non-trivial inter-process communication. Some refer to this type of problems as

communication-intensive. For data-parallel problems, the requirement for communication is determined by data dependency among processes.

Function interpolation is an example of embarrassingly parallel problems. The goal in this problem is to find a proper linear combination of a known set of basis functions $\mathrm{span}\{\phi_1, \dots, \phi_n\}$ in a certain finite function space $\mathcal{V}$, such that the resulting function $f = \sum_{j=1}^{n} c_j \, \phi_j(x)$ is equivalent to or approximates a given target function g. The solution to this problem is to compute all the coefficients $\{c_1, \dots, c_n\}$, which can be obtained by evaluating the target function g at each point in $\mathcal{V}$ associated with a basis function. This problem can be decomposed into n subtasks of each computing a coefficient. There is no data dependency between these subtasks, therefore, they can be executed concurrently or asynchronously, in any order. The only place in the program that requires communication is in the end, where results from all processes are aggregated.

A computational fluid dynamics (CFD) simulation is a typical example of communication-intensive problems. The simulation domain is partitioned into a number of subdomains, upon each of which computation is carried out by a process. The simulation advances in time. The computation of one time step of a subdomain requires information from its adjacent subdomains, which means, in each time step, a process must communicate with its corresponding neighbor processes. Inter-process communication for such problems is intensive in terms of message size and frequency. It is much more difficult to incorporate resource elasticity into such problems, because when there are changes in resources, data must be redistributed carefully such that a correct communication topology is preserved.

In the following sections, we discuss Elastic MPI programming models for communication-intensive and embarrassingly parallel problems.

4.2 SPMD with Single Computation Phase

SPMD refers to the execution model in which the same program is executed on multiple data sets. In the parallel context, it means every process performs the same operations to its local data. This model is widely used among MPI applications.

In communication-intensive problems such as grid-based simulations, the computation domain is decomposed and distributed among processes. Processes perform the same or similar operations to their local data and communicate with one another periodically. Their computation-communication cycles must be synchronized, otherwise performance would suffer due to large communication overhead from waiting for messages. The SPMD model is suitable for such problems that require frequent communication and synchronization.

Algorithm 4.1 presents a schematic MPI program with a single computation loop implementing the SPMD model. This program has a typical workflow of an HPC application: starting with certain initialization steps, it does the major computation in a loop, then it finalizes and terminates. We can divide such a workflow into 3 phases: *initialization*, *computation*, and *finalization*. The distinction of phases is practical in parallel programming, because these phases have different scalability and we normally parallelize only the computation phase. In Elastic MPI, resource adaptation is also only considered for the computation phases, where there are heavy workloads and require the most computing power. Algorithm 4.2 presents an Elastic MPI version of Algorithm 4.1. In it resource adaptivity is introduced to the main loop.

For programming with Elastic MPI, the following must be considered:

Algorithm 4.1: MPI program in SPMD with single computation phase

```
1  Function Main():
2      MPI_Init()
3      Initialize numRanks, rank, etc.
       // 1. Initialization
4      Initialize computation data, perform domain decomposition, etc.
5      t ← 0
       // 2. Computation: main loop
6      while t < T_max do
7          Perform communication
8          Perform computation
9          Produce output
10         t ← t + Δt
11     end
       // 3. Finalization
12     MPI_Finalize()
13 End
```

- frequency of resource adaptation,
- data repartition and redistribution for resource changes,
- data migration from `LEAVING` processes,
- integration of `JOINING` processes with the preexisting processes.

The next paragraphs explain Algorithm 4.2 in detail with the above considerations.

Frequency of Resource Adaptation

The frequency for resource adaptation is application dependent. It can range from as frequent as once every iteration to never, in which case it is basically a static MPI application. The ideal frequency should reflect all resource change decisions from the resource manager yet result in a small amount of accumulative adaptation overhead.

On one hand, resource changes do not occur at arbitrary time. The resource manager collects performance data and makes decisions periodically. Therefore, it is unnecessary to probe the resource manager much more frequently than it makes decisions. On the other hand, if an application probes too infrequently, it may miss many real-time decisions and hinder resource efficiency.

A third consideration for the adaptation frequency is the application's inherent load balancing scheme. Dynamic applications require periodic load balancing operations to mitigate imbalances caused by their dynamic workload behavior. Every time resource changes, applications must also performance data redistribution operations. If load balancing due to resource changes could overlap with the application's built-in load balancing operations, the adaption overhead would be significantly reduced.

Algorithm 4.2: Elastic MPI program in SPMD with single computation phase

```
1  Function Main():
2      MPI_Init_adapt()                          // returns process status in procStatus
3      Initialize numRanks, rank, etc.
       // 1. Initialization: JOINING processes bypass
4      if procStatus = JOINING then
5          Resource_Adapt()                      // Loop counter t is synced during adapt
6      else
7          Initialize computation data, perform domain decomposition, etc.
8          t ← 0
9      end
       // 2. Computation: main loop
10     while t < T_max do
11         Perform communication
12         Perform computation
13         Produce output
14         t ← t + Δt
15         if Time to probe Resource Manager then
16             MPI_Probe_adapt()                 // returns adapt decision in adaptFlag
17             if adaptFlag says to adapt then
18                 Resource_Adapt()
19             end
20         end
21     end
       // 3. Finalization
22     MPI_Finalize()
23 End

24 Function Resource_Adapt():
25     MPI_Comm_adapt_begin()
26     if there are LEAVING processes then
27         Save or transfer data from LEAVING processes
28     end
29     if there are JOINING processes then
30         JOINING processes initialize necessary data objects
31         Synchronize variables (loop counter, etc.) among the new resource group
32     end
33     Redistribute computation data among the new resource group
34     Sync data (loop counters, etc.)
35     MPI_Comm_adapt_commit()                   // MPI_COMM_WORLD is updated
36     Update numRanks, rank, procStatus, etc.
37 End
```

Resource Adaptation & Data Redistribution

In the main loop, `MPI_Probe_adapt` returns a signal to indicate whether or not there are resource changes. If the signal is positive, the application enters the `Resource_Adapt` function, which opens an adaptation window. The adaptation window is a transition period in which both current and new resources are accessible. The application must perform necessary data migration and synchronization within the window.

`MPI_Comm_adapt_begin` provides information on the resource changes, such as the number of `STAYING`, `JOINING` and `LEAVING` processes, as well as temporary communicators to assist data migration and synchronization. Upon commit, these temporary communicators would be destroyed, and `MPI_COMM_WORLD` would be updated, i.e., `LEAVING` processes are excluded and no longer accessible, and `JOINING` processes are included. After committing, it is necessary to also update all `MPI_COMM_WORLD` dependent variables such as the rank of each process and the total number of ranks.

For data redistribution, the shown steps in the adaptation window can handle not only the pure reduction and expansion cases, but also the more general case (though not yet supported) of having both `LEAVING` and `JOINING` processes in one adaptation. First, data integrity is regained by saving or transferring the partitions residing in the `LEAVING` processes (if exist). Then, the `JOINING` processes (if exist) are prepared to receive data partitions. Finally, data is redistributed among the new set of resources, i.e., the union of `STAYING` and `JOINING` processes. Redistribution can be done via point-to-point communications, collective reduce-broadcast communications, MPI I/O, or however is most suitable for the problem.

Integration of `JOINING` Processes

While handling of the `LEAVING` processes is straight forward, i.e., saving their data partitions to the file system or transferring their data to the `STAYING` processes, handling of the `JOINING` processes requires more than data preparation.

Creation of the `JOINING` processes with complete memory copy from the preexisting processes is blocking, and the copy operation can take up a long time depending on the application's memory usage. Therefore, by design, `JOINING` processes are created with empty memory spaces and function stacks. They must start execution from the beginning and go through the same procedures to achieve a similar memory state as the preexisting processes. However, except for the initialization of some necessary variables and objects, `JOINING` processes should avoid long, complex initial operations such as domain initialization, initial domain decomposition and distribution, because they would receive proper data preparation inside the `Resource_Adapt` block. This is reflected in Algorithm 4.2 lines 4 to 9, where the `JOINING` processes are identified and routed immediately to enter the `Resource_Adapt` function, bypassing domain data initialization.

Preexisting processes are executing the computation loop during the creation of `JOINING` processes. They are notified only when the `JOINING` processes are ready. `JOINING` processes would reach the `Resource_Adapt` block first. Then, the preexisting processes would also enter the `Resource_Adapt` block at the end of an iteration (line 17).

After returning from `Resource_Adapt`, the preexisting processes start a new iteration from line 10. The `JOINING` processes, on the other hand, will also reach the same line after returning from their `Resource_Adapt` call. Of course, both groups should have syn-

chronized the loop counter t in the resource adaptation window. At this point, `JOINING` processes are successfully integrated with the preexisting processes.

4.3 SPMD with Multiple Computation Phases

Extending from the case with a single computation phase, we now consider an SPMD type application with multiple computation phases, with each phase consisting of a loop. If all computation loops are made elastic, then a problem arises: how do the `JOINING` processes know which loop to enter? Algorithm 4.3 shows a solution to this problem with the introduction of an additional parameter – phaseID.

In this example, there are two computation loops, each of which is wrapped in a conditional block, which is only executed if the phaseID matches a given value. As for the preexisting processes, they enter the program, go through the original initialization phase, and get phaseID initialized to 1 (line 8). Then they enter the first `if`-block and execute the first loop. Upon completion, their phaseID is updated to 2, so they continue on to the second `if`-block and execute the second loop.

The `JOINING` processes are directed immediately to the `Resource_Adapt` block, which is similar to the one shown in Algorithm 4.2 and is not listed in this section again. In the adaptation block, they are prepared with computation data, updated loop counters t, i and phaseID. This allows them to identify the correct computation loop and join in at the correct iteration.

4.4 Master-Worker with Single Computation Phase

The master-worker model refers to the parallel execution scheme in which a master process divides a task into a number of subtasks (jobs), dispatches them to several worker processes and aggregates results; meanwhile, the worker processes compute received jobs one after another and send back results until they receive a stop signal.

Many embarrassingly parallel problems implement this model. Due to the absence of data dependency in these problems, data decomposition can be done arbitrarily and synchronization is not required. Therefore, data can usually be divided into equal-load chunks and computed asynchronously. Processes might work at different speeds, due to different CPU clock frequencies, temperatures, hardware conditions, etc.. The flexibility and asynchronism in this scheme allow processes to work more efficiently at their own pace.

Algorithm 4.4 shows a schematic MPI program implementing the master-worker model. Note that a master-worker implementation does not imply embarrassingly parallelism, and vice versa. Indeed, embarrassingly parallel problems can implement other schemes such as SPMD, and the master-worker model can also be employed by other problem types.

Resource adaptivity can be incorporated in the master-worker scheme with some extra care. Theoretically, processes can be added and removed freely without affecting other processes because there is no data dependency. However, the resource adapt operations (`MPI_Comm_adapt_begin` and `MPI_Comm_adapt_commit`) are collective, which means the master process must interrupt the computation, inform and prepare the workers for adaptation. Algorithms 4.5 and 4.6 shows an Elastic MPI program implementing the master-worker model.

Algorithm 4.3: Elastic MPI program in SPMD with multiple computation phases

```
1  Function Main():
2      MPI_Init_adapt()                          // returns process status in procStatus
3      Initialize numRanks, rank, etc.
       // 1. Initialization: JOINING processes bypass
4      if procStatus = JOINING then
5          Resource_Adapt()                      // phaseID and loop counters t, i are synced
6      else
7          Initialize computation data, perform domain decomposition, etc.
8          t ← 0, i ← 1, phaseID ← 1
9      end
       // 2. Computation: first loop
10     if phaseID = 1 then
11         while t < T_max do
12             Perform communication
13             Perform computation
14             Produce output
15             t ← t + Δt
16             if Time to probe Resource Manager then
17                 MPI_Probe_adapt()             // returns adapt decision in adaptFlag
18                 if adaptFlag says to adapt then
19                     Resource_Adapt()
20                 end
21             end
22         end
23         phaseID ← 2
24     end
       // 2. Computation: second loop
25     if phaseID = 2 then
26         for i do
27             Perform communication
28             Perform computation
29             Produce output
30             if Time to probe Resource Manager then
31                 MPI_Probe_adapt()             // returns adapt decision in adaptFlag
32                 if adaptFlag says to adapt then
33                     Resource_Adapt()
34                 end
35             end
36         end
37     end
       // 3. Finalization
38     MPI_Finalize()
39 End
```

Algorithm 4.4: MPI program in master-worker with single computation phase

```
1  Function Main():
2      MPI_Init()
3      Initialize numRanks, rank, etc.
       // 1. Initialization
4      Initialize computation data
       // 2. Computation: single master-worker routine
5      if is Root then
6          Master()
7      else
8          Worker()
9      end
       // 3. Finalization
10     MPI_Finalize()
11 End

12 Function Master():
13     Initialize jobsArray              // Track job status: Todo, Progress, or Done
14     Seed workers
15     while not all jobs are Done do
16         Receive a job result from any worker
17         if there are Todo jobs then
18             Send a Todo job to the same worker
19         end
20         Aggregate the job result
21     end
22     Send terminate signal to all workers
23 End

24 Function Worker():
25     while true do
           // Message body carries job data. Message status object encodes instruction.
26         Receive a message from Master
27         Extract Master's instruction from MPI status object
28         if instruction is to terminate then
29             break
30         end
31         if instruction is to work then
32             Compute the job
33             Send job result to Master
34         end
35     end
36 End
```

Algorithm 4.5: Elastic MPI program in master-worker with single computation phase, Main function

```
1  Function Main():
2      MPI_Init_adapt()                          // returns process status in procStatus
3      Initialize numRanks, rank, etc.
       // 1. Initialization: JOINING processes bypass
4      if procStatus = JOINING then
5          Resource_Adapt()
6      else
7          Initialize computation data
8      end
       // 2. Computation: single master-worker routine
9      if is Root then
10         Master()
11     else
12         Worker()
13     end
       // 3. Finalization
14     MPI_Finalize()
15 End
```

The Main function

Redirecting the `JOINING` processes is a crucial step in Elastic MPI programming. In the elastic main function in Algorithm 4.5, a conditional block is added in the beginning to identify and route the `JOINING` processes to the `Resource_Adapt` block immediately. This is necessary because without redirection, the `JOINING` processes would enter the worker function directly and wait for instructions, while the `Master` and other preexisting workers are in the adaptation block waiting for the `JOINING` processes, thus creating a so-called *deadlock* situation.

A very subtle yet important thing to be aware of in Elastic MPI is that identifying the `Root` process requires not only the rank but also the process status. This is because when the `JOINING` processes are created and not yet integrated, i.e., before calling `MPI_Comm_adapt_commit`, they are in a separate process group with their own world communicator and corresponding ranks starting from 0. Therefore, a filter on only the rank equals to 0 is inefficient, because it would result in two processes – the preexisting Rank 0 and the joining Rank 0. Note that choosing the `Root` to be the `Master` process is not mandatory.

The Master Process

In the master function in Algorithm 4.6, a job array is used for keeping track of job status. Each element of the job array corresponds to a job, and the value of the element corresponds to the job status. A worker array is used in a similar manner for keeping track of the worker status, which is a requirement for the `Master`, because when it interrupts

Algorithm 4.6: Elastic MPI program in master-worker with single computation phase, Master & Worker functions (continue from Algorithm 4.5)

```
16 Function Master():
17 |  Initialize jobsArray                  // Track job status: Todo, Progress, or Done
18 |  Initialize workersArray                     // Track worker status: Active or Idle
19 |  Seed workers
20 |  while not all jobs are Done do
21 |  |  if there are Active workers then
22 |  |  |  Receive a job result from any worker
23 |  |  |  Aggregate job result
24 |  |  end
25 |  |  if there are Todo jobs and there are Idle workers then
26 |  |  |  Send a Todo job to an Idle worker
27 |  |  end
   |  |  // Only make sense to adapt if there are more Todo jobs
28 |  |  if Time to probe Resource Manager then
29 |  |  |  MPI_Probe_adapt()                 // returns adapt decision in adaptFlag
30 |  |  |  if adaptFlag says to adapt and there are Todo jobs then
31 |  |  |  |  Check all workers status, collect job results from Active workers
32 |  |  |  |  Send adapt signal to all workers
33 |  |  |  |  Resource_Adapt()
34 |  |  |  |  Resize and update workersArray                      // workers changed
35 |  |  |  |  Seed workers
36 |  |  |  end
37 |  |  end
38 |  end
39 |  Send terminate signal to all workers
40 End

41 Function Worker():
42 |  while true do
   |  |  // Message body carries job data. Message status object encodes instruction.
43 |  |  Receive a message from Master
44 |  |  Extract Master's instruction from MPI status object
45 |  |  if instruction is to terminate then
46 |  |  |  break
47 |  |  end
48 |  |  if instruction is to adapt then
49 |  |  |  MPI_Probe_adapt()           // workers need to know its process status
50 |  |  |  Resource_Adapt()
51 |  |  end
52 |  |  if instruction is to work then
53 |  |  |  Compute the job
54 |  |  |  Send job results to Master
55 |  |  end
56 |  end
57 End
```

computation for resource adaptation, it needs to know which workers are still active such that it can try to collect results from them.

In the while loop, message receive and send are protected by conditionals. The `Master` only receives a message, if it knows for sure there are messages being sent to it, i.e., there are active workers which means job results will be sent when they complete their jobs. It only dispatches a job, if there are jobs to do and there are idling workers.

The `Master` uses a counter to control the frequency of probing the resource manager. Only when there are resource changes and there are more jobs left to do, it invokes a sequence of operations to interrupt worker computation for resource adaptation.

The `Master` must first ensure that all workers have finished their current jobs by collecting results from active workers. It then sends out the adapt signal to workers and enters the `Resource_Adapt` block itself. Post adaptation, the `Master` updates the worker array both in size and values to reflect the new resources. To get them back in computation, the `Master` must seed the workers again.

Worker Processes

The worker function in Algorithm 4.6 is relatively straight forward. Workers wait for the instruction from the `Master` in a while loop and take actions accordingly. There are three possible actions: terminate computation by exiting the loop, resource adapt by entering the `Resource_Adapt` function, and compute a job. For resource adaptation, workers must first get their process status by calling `MPI_Probe_adapt`. The `Master`'s instruction is encoded in the status object returned by the MPI receive function (this status object is different than the process status). A worker receives a message from the `Master`, extract the instruction, and decide what to do. The body of the message carries job data if the instruction is to compute, otherwise, it carries dummy values which would be ignored.

The `Resource_Adapt` Function

The `Resource_Adapt` function is similar to the one shown in Algorithm 4.2. To enter this function collectively, the `Master` must interrupt worker computation and send workers the adapt signal, which might introduce considerable overhead. For embarrassingly parallel problems, there is no need for data migration from `LEAVING` processes, nor data repartition and redistribution inside the adaptation window. The only thing needed is data preparation for the `JOINING` processes, such that they can carry out computation on jobs.

4.5 Master-Worker with Multiple Computation Phases

Let's consider a surrogate model[1] construction problem: a surrogate model is initially generated by a certain algorithm. The accuracy of the surrogate model can be evaluated with some error indicator. If the surrogate is not accurate enough, it should be refined repeatedly until its error indicator satisfies a certain threshold. Algorithm 4.7 demonstrates the Elastic MPI implementation of such problem.

[1] A surrogate model is an approximation to a high-fidelity model that is complex and computationally expensive. A surrogate model is reduced in complexity, and thus, is computationally inexpensive. It should produce similar results as the high-fidelity model with tolerable errors.

Algorithm 4.7: Elastic MPI program in master-worker with multiple computation phases, Main function

```
1  Function Main():
2      MPI_Init_adapt()                          // returns process status in procStatus
3      Initialize numRanks, rank, etc.
       // 1. Initialization: JOINING processes bypass
4      if procStatus = JOINING then
5          Resource_Adapt()                       // loop counter err is synced
6      else
7          Initialize surrogate model
8          Initialize surrogate model error err
9      end
       // 2. Computation: multiple master-worker routines
10     while err > Tol do
11         if is Root then                        // Refine surrogate model
12             Master()
13         else
14             Worker()
15         end
16         Master update err, broadcast to workers
17     end
       // 3. Finalization
18     MPI_Finalize()
19 End
```

Each refinement of the surrogate is carried by a resource-adaptive master-worker routine, in which multiple resource adaptations may occur depending on the workload of the refinement. The master and worker functions are similar to those shown in Algorithm 4.6, and the `Resource_Adapt` function is similar to the one shown in Algorithm 4.2, hence, they are not listed again in this section.

Since the number of model refinement is indefinite, the master-worker routine must be wrapped in a loop, which terminates only when the desired model accuracy is reached. This program contains multiple resource-adaptive computation phases, with each phase being an iteration of the while loop.

The key to handle multiple computation phases is the phase identifier, which must be synchronized between the preexisting and `JOINING` processes during the resource adaptation window. In a program with a definite number of phases that are wrapped in conditional blocks, such as Algorithm 4.3, a phase identifier can help to determine which block is being executed. In a program with computation phases wrapped in a loop, such as Algorithm 4.7, the loop counter helps to identify the phases and must be synchronized.

As discussed in Section 3.4.2, in the current implementation, the master node (containing Rank 0) is persistent. This means that the `Master` process, if pinned to Rank 0 or any rank resides in the master node, is assumed unchanged throughout program execution. There is currently no fault-tolerant mechanism to handle failure of the `Master` process (or the master node).

4.6 Summary

In this chapter, we discussed two types of parallel applications classified based on communication requirements: embarrassingly parallel and communication-intensive.

We presented the Elastic MPI implementation of two parallel programming schemes: SPMD and master-worker, which are suitable for the communication-intensive and embarrassingly parallel problems respectively. Besides handing data migration and repartitioning in the adaptation window, routing the `JOINING` processes such that they merge with the preexisting processes at a correct place is a crucial step in both schemes.

We also discussed solutions for adding resource adaptivity to multiple computation phases. The key is to introduce a variable to distinguish each phase and have it synchronized between the preexisting and `JOINING` processes in the adaptation window.

PART III

RESOURCE-AWARE AND ELASTIC PARALLEL SOFTWARE DEVELOPMENT

5 Elastic Parallel Tsunami Simulation with Adaptive Mesh Refinement

Based on the requirements for communication and synchronization, parallel applications can be categorized as *embarrassingly parallel* or *communication-intensive*, with the latter being a more common type among HPC applications.

The simulation of tsunami wave propagation is a communication-intensive problem. It involves solving a system of time-dependent Partial Differential Equations (PDEs). Analytic solutions usually do not exist due to the complexity of the PDE system, therefore, numerical methods are required. The simulation domain (the ocean) is typically discretized with a grid (also known as mesh), and the unknowns of the PDE system are placed at grid elements such as grid cells, vertices or edges. Solving the PDE system means computing the values of unknowns at each grid element. When the simulation is parallelized, the grid is partitioned and distributed among processes.

The computation of unknowns at each grid element is dependent on the values from the neighboring elements, which means, for each process, the computation of values in its boundary elements requires data from other processes that contain their neighboring elements. Each process must have knowledge on the domain distribution (at least partially) such that it knows with which processes to communicate. Such data dependency between parallel processes poses challenges in realizing resource elasticity, because it requires preservation of a definite communication topology in spite of frequent domain repartitioning and redistribution due to resource changes.

To investigate the feasibility and practicality of the Elastic MPI framework for communication-intensive parallel applications, we selected a representative large-scale tsunami simulation for experimentation. sam(oa)^2 , which stands for *Space-filling Curves and Adaptive Meshes for Oceanic and Other Applications*, is a software developed by Meister [77] for simulations based on AMR. A tsunami simulation based on sam(oa)^2 is selected for the following reasons:

1. It represents a large population of HPC applications that are large-scale and communication-intensive. It implements a popular parallel execution model SPMD.
2. This application has a very dynamic computational workload, hence it has a demand for runtime resource adjustment. Moreover, the application is compute-bound, suggesting that it is easier to scale.
3. With AMR, the underpinning grid of the simulation varies constantly. For that, the application has a built-in data redistribution and load balancing scheme to handle

frequent workload changes and imbalances. These functionalities can be used to handle data redistribution required by resource changes.

In this chapter, we first discuss the foundation which the spatial discretization scheme of sam(oa)^2 is implemented. Then, we examine the implementation of the tsunami simulation in sam(oa)^2 . Following this, we apply a transformation to the application using the Elastic MPI library. Lastly, we conduct performance tests to determine the impact of introducing runtime resource adaptivity.

5.1 Sierpiński Space-filling Curves

To truly understand the dynamic nature of the adaptive mesh underpinning the tsunami simulation in sam(oa)^2, we must first discuss the foundational concept upon which sam(oa)^2 is built – Space-filling Curves (SFCs).

Motivated by Georg Cantor's earlier counter intuitive result that there exists a one-to-one correspondence between points on a unit line segment and points in a d-dimensional space [78], and with the intention to construct a continuous mapping from the unit interval onto the unit square, Giuseppe Peano discovered the first SFC in 1890 [79], which is now called the Peano curve. Since then more SFCs were discovered over the years, such as the Hilbert curve [80] and the Lebesgue curve [81].

Let γ be a mapping from a one-dimensional to a d-dimensional space, e.g.,

$$\gamma : [0,1] \to [0,1]^d.$$

Note that the domain and codomain of γ are not limited to unit spaces. γ is considered *space-filling* if it is surjective, that is, for each point $(x_1, \ldots, x_d) \in [0,1]^d$, there exists a point $k \in [0,1]$ such that $\gamma(k) = (x_1, \ldots, x_d)$. The image of such a surjective mapping is said to be a SFC if it is also continuous [82, 83].

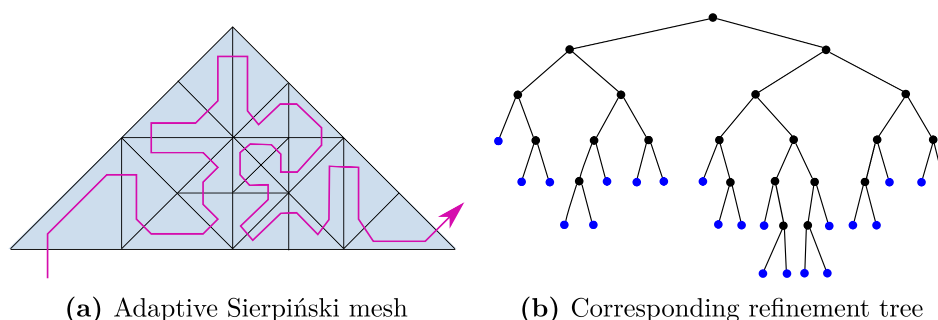
A common application of SFCs is for mapping multi-dimensional data to one dimension, such that one-dimensional access methods can be exploited. In computer simulation, they are used for adaptive grid generation with a 1-D ordering of the grid elements.

The Sierpiński Curve

The adaptive meshes implemented in sam(oa)^2 are based on the Sierpiński SFC, which was discovered by Waclaw Sierpiński in 1912 [84]. It is a mapping from a 1-D to 2-D domain created by recursive substructuring of right, isosceles triangles.

Meshes generated by square-based SFCs, such as Peano curves, are known to have the problem of being non-conforming (having hanging nodes) when adaptively refined. However, for triangle-based meshes, this problem can be resolved by using *newest vertex bisection* for adaptive refinement [85, 86]. The combination of Sierpiński curve traversal and newest vertex bisection results in a recursive traversal algorithm for structured, conforming and adaptive triangular grids [87].

Figure 5.1 illustrates the creation of an adaptive mesh using Sierpiński curve traversal with newest vertex bisection and its corresponding refinement tree. In simulation problems, data are associated with grid cells, vertices and edges. The curve in Figure 5.1(a) represents the order in which data are accessed during computation. Accessing from the first to the last grid element along the curve is called a Sierpiński traversal, with which grid data



(a) Adaptive Sierpiński mesh (b) Corresponding refinement tree

Figure 5.1: Creation of an adaptive triangular mesh based on Sierpiński curve traversal with newest vertex bisection, and the corresponding refinement tree. Data are associated with grid cells, vertices and edges, and transformed into a 1-D representation by the Sierpiński curve. The algorithm accesses data via grid traversal as a whole. Random access to individual elements is not permitted. The refinement tree represents the hierarchical structure of the grid. Nodes are accessed by depth-first traversal from left to right. Only the *leave nodes* (nodes in blue) correspond to actual grid cells.

are accessed and processed as a whole. Random access to individual element data is not permitted.

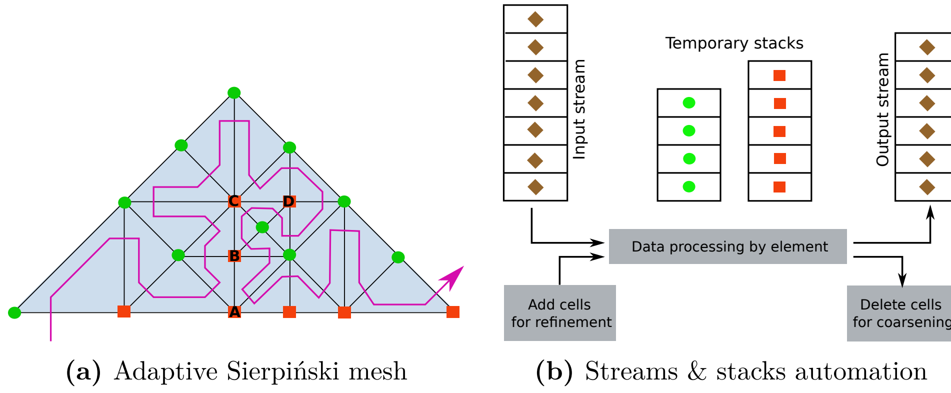
The refinement tree in Figure 5.1(b) represents the hierarchical structure of the grid. Nodes are accessed in a depth-first traversal order from left-to-right of the tree. Only the *leaf nodes*, i.e., nodes with no children, colored blue in Figure 5.1(b), correspond to actual grid cells and are processed during a Sierpiński traversal. In other words, the left-to-right order of all leaf nodes is the order of grid cells being accessed during a traversal.

Cache-Oblivious Sierpiński Traversal in sam(oa)^2

Due to the one-dimensional representation, grid data can be stored in structures with contiguous memory spaces such as streams and stacks, enabling cache-oblivious data access pattern. sam(oa)^2 implements such a scheme by storing and accessing data purely on streams and stacks [88, 89].

Figure 5.2 demonstrates with vertex data how the cache-oblivious data access scheme is achieved in sam(oa)^2. A complete traversal includes two passes: a forward and a reverse pass. The Sierpiński curve shown in Figure 5.2(a) represents the forward traversal. Based on their location relative to the forward curve, i.e., on the left or right side of the curve, vertices are colored red or green respectively. The red vertices A, B, C and D, for instance, are first accessed in-order during the forward traversal, and then in reverse order during the reverse traversal.

For computation, the vertex data are initially stored in an input data stream, as shown in Figure 5.2(b). They are then processed and pushed onto the green and red stacks in the order they are encountered during the forward traversal. Then they are operated on again in reverse order during the reverse traversal, and the results are stored in an output stream. Similar procedures are also applied on data associated to cells and edges.



(a) Adaptive Sierpiński mesh **(b)** Streams & stacks automation

Figure 5.2: Demonstration of a pure streams- and stacks-based data storage and access scheme with vertex data as an example. Vertices are colored green and red based on their location relative to the forward Sierpiński curve. Vertex data are initially stored in an input stream and pushed to separate (green and red) stacks in the order they are encountered during the forward traversal. They are then processed in reverse order during reverse traversal and the results are stored in an output stream.

5.2 Tsunami Simulation in sam(oa)^2

This section presents the theory and implementation details of the tsunami simulation in sam(oa)^2 , which is based on the work of Meister [77].

5.2.1 Modeling the Tsunami Wave Propagation

For a tsunami simulation, we consider a three-dimensional domain which includes all or part of an ocean. Typically, the wavelength of a tsunami largely exceeds the water height, as the wavelength stretching in the two horizontal dimensions reaches hundreds of kilometers, while the ocean depth in the vertical dimension is on the scale of a few kilometers. It is the high wavelength-to-water height ratio that classifies the domain in which the tsunami occurs as *shallow water*, and so the phenomenon can be modeled by the Shallow Water Equations (SWE), which allow an approximation that integrates over the vertical axis while ignoring vertical derivatives [90]. This approach models water to form a vertical pillar in each 2D position $\vec{x} = (x, y)$ with space- and time-dependent water height $h(\vec{x}, t)$ and velocity $\vec{u}(\vec{x}, t) = (u, v)$, transforming the simulation domain into a 2-D domain.

Of the different variations of SWE, the implementation of the tsunami simulation in sam(oa)^2 is based on the following equations,

$$\begin{aligned} h_t + (hu)_x + (hv)_y &= 0 \\ (hu)_t + (hu^2 + \tfrac{1}{2}gh^2)_x + (huv)_y &= -ghb_x \\ (hv)_t + (huv)_x + (hv^2 + \tfrac{1}{2}gh^2)_y + &= -ghb_y. \end{aligned} \tag{5.1}$$

This hyperbolic system describes the conservation of mass and momentum. Mass is represented by the water height h, because mass equals density times volume, i.e.,

$$m = \rho \cdot \Delta A \cdot h,$$

where ρ denotes water density and ΔA represents the infinitesimal area at each 2-D position, and the constants ρ and ΔA can be canceled out. For the same reason, momentum in the two horizontal dimensions are represented by hu and hv. g represents the gravitational acceleration and b represents the bathymetric data (height of the ocean floor relative to Earth's geoid).

System (5.1) can be expressed in the general form of a balance law as

$$\mathbf{q}_t + \mathbf{f}(\mathbf{q})_x + \mathbf{g}(\mathbf{q})_y = \mathbf{\Psi}(\mathbf{x}), \tag{5.2}$$

where

$$\mathbf{q} := \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \quad \mathbf{f}(\mathbf{q}) := \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}, \quad \mathbf{g}(\mathbf{q}) := \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, \quad \mathbf{\Psi}(\mathbf{x}) := \begin{bmatrix} 0 \\ -ghb_x \\ -ghb_y \end{bmatrix}.$$

Systems of such form as in (5.2) are well studied. There exist several methods for solving for the unknown $\mathbf{q}$, especially for SWE [91, 92, 93, 94].

5.2.2 Finite Volume Discretization and Explicit Time Stepping

For spatial discretization, the tsunami simulation in sam(oa)^2 adopts the finite volume approach by [95], in which unknowns are represented by cell-average values in each cell element. Let

$$\mathbf{q}_j^{(t)} := \begin{bmatrix} h_j \\ h_j u_j \\ h_j v_j \end{bmatrix}$$

be the corresponding unknown in cell j at time t, and

$$\mathcal{F}(\mathbf{q}_j^{(t)}, \mathbf{q}_i^{(t)})$$

the transport of mass and momentum (the so-called *net update*) into and out of cell j from an adjacent cell i at time t. An explicit Euler update scheme can then be written as

$$\mathbf{q}_j^{(t+\Delta t)} = \mathbf{q}_j^{(t)} + \frac{\Delta t}{V_j} \sum_{i \in \mathcal{N}(j)} A_{j,i}\, \mathcal{F}(\mathbf{q}_j^{(t)}, \mathbf{q}_i^{(t)}), \tag{5.3}$$

where V_j represents the volume of cell j, $A_{j,i}$ represents the intersection area between cell j and cell i, and $\mathcal{N}(j)$ denotes the set of all neighboring cells of j. Equation (5.3) can be understood as the updated values in cell j at time $t + \Delta t$ being calculated by the values at time t plus the flux changes during the time interval Δt from all neighboring cells.

The most important component for the time stepping scheme in finite volume methods is the *flux solver*, which computes the net updates $\mathcal{F}(\mathbf{q}_j^{(t)}, \mathbf{q}_i^{(t)})$ between adjacent cell pairs. There exist different flux solvers with trade-offs between accuracy and computational complexity. While sam(oa)^2 implements several flux solvers, we skip over their formulation and implementations details (which can be found in [77]) and treat them as black box solvers.

As for the refinement indicator, a relative criterion is chosen as

$$\left| \frac{h_j^{(t+\Delta t)} - h_j^{(t)}}{\Delta t} \right| V(\Omega_j) > \mathrm{Tol}_h \, V(\Omega_{\min}), \tag{5.4}$$

where $\mathrm{Tol}_h > 0$ is an arbitrarily chosen constant. The left-hand-side of the inequality sign translates to the flux divergence. This criterion indicates refining cells with strong flux changes and merging cells with the same water height. With such an indicator, grid refinement will occur mostly at wave fronts and coarsening will occur mainly in areas where the ocean is at rest.

5.2.3 Sierpiński Curve for Parallelization

The convenient 1D representation of grid cells, vertices and edges provided by the Sierpiński order facilitates parallelization of 2D grids. In addition, the *Hölder-continuity* of Sierpiński curves ensures well-formed partitions that are edge connected [96], which resolves the known NP-hard problem of finding uniform partitions for adaptive meshes while minimizing communication [97]. By adopting the straight forward approach of cutting the grid into partitions of uniform computational load along the Sierpiński curve, sam(oa)^2 achieves a good load-balanced parallelization scheme with minimized inter-process communication.

Computational load in sam(oa)^2 is modeled as an abstract cost function, which is defined as either a weighted sum of the number of grid cells, vertices and edges, or an estimate for the execution time based on runtime measurements. Once the computational load is defined, sam(oa)^2 divides the grid into *sections* with uniform load at best effort. Sections are independent computational units that consist of the grid elements (cells, vertices and edges) and their associated data corresponding to contiguous intervals of the Sierpiński curve. The union of all sections forms the grid.

sam(oa)^2 supports shared-memory, distributed-memory or hybrid parallelization with OpenMP and MPI. It implements different controlling mechanisms to define number and size of sections and how to distribute them to processors, e.g., a distributed algorithm assigns sections as atomic units to processors, a node-local algorithm decides how many sections are to be created and how large they are. For hybrid type parallelization, if sections are set to be atomic, i.e., splitting of sections is not permitted, a multiple of n sections are assigned to a MPI process, where n is the number of physical processors associated with a process. In this case, processes first compute their local sections and mark grid cells for refinement or coarsening. An estimate of the new computational load of each section is then computed, and sections are distributed based on their new loads. Grid refinement and coarsening is performed locally only after the load balancing (distribution of sections) step, followed by a local repartitioning step of sections. In another case where splitting of sections are allowed, processes would perform actual grid refinement before redistributing sections. After load balancing, sections will also be repartitioned again locally. Besides different section control schemes, in pursuit of more optimal load balancing results, sam(oa)^2 also implements other advanced features such as thread-level work stealing and different cost function models that are suitable for different situations.

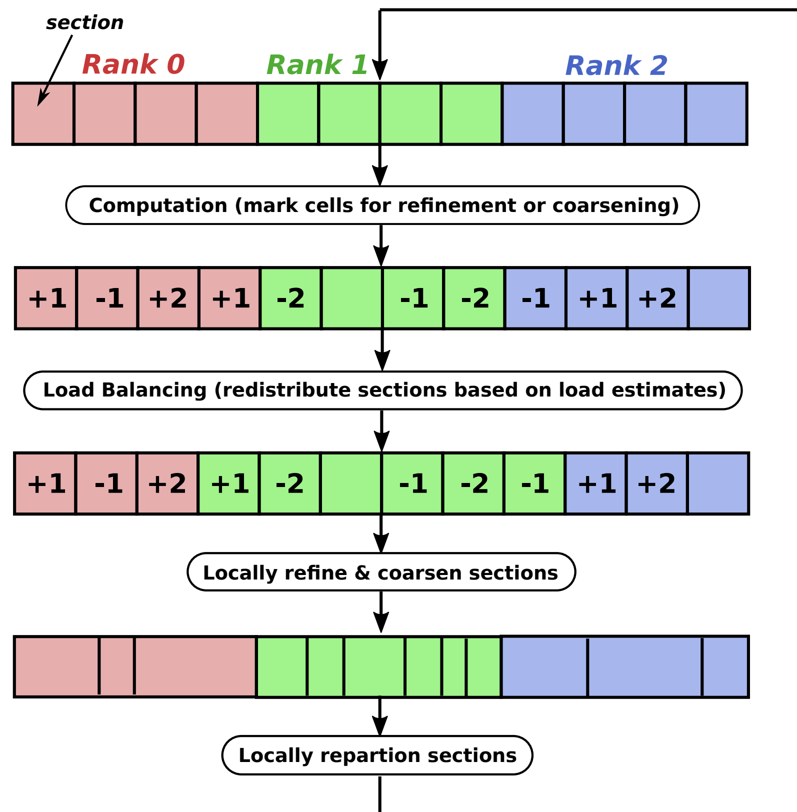


Figure 5.3: Demonstration of parallelization, adaptive refinement and load balancing steps with the atomic section approach. A computation step is first performed and cells are marked for refinement or coarsening. Sections are distributed across processes based on their load estimates. The actual grid refinement and coarsening step is performed after the load balancing step. Lastly, sections are repartitioned locally.

Load Balancing with Chains-on-chains Restriction

Dynamic grid refinement imposes a restriction on grid partitioning, i.e., grid cell pairs that are feasible for coarsening (being merged into one cell) should be allocated in such a way that doing so is possible. In other words, any cell pair that can be merged in a coarsening operation should be assigned to the same or consecutive processes. When we take sections as independent computational units, this restriction should also be reflected on the section-level, i.e., sections must be located on the same process or on neighboring processes if they are in consecutive Sierpiński order. A direct solution is to enumerate the processes by consecutive integers, sort all the sections in Sierpiński order and assign them to processes with a non-decreasing mapping. The resulting 1-D load balancing scheme is known as the *chains-on-chains partitioning* scheme [98, 99].

Parallelization, adaptive refinement and load balancing (distribution of sections) steps are demonstrated in Figure 5.3 in the atomic section parallelization approach. An array of rectangles, representing a number of sections in the 1-D Sierpiński order, are colored in red, green and blue to indicate the processes in which the sections reside. At the beginning of the iteration, a computation step is performed and based on the updated data, cells are marked for refinement or coarsening and a load estimate of each section is computed. Then a load balancing step is performed by distributing sections in a chains-on-chains manner across processes based on their load estimates. Afterwards, an actual grid refinement step is performed locally by each process followed by a step of section repartitioning.

5.2.4 Main Simulation Steps

Major steps of the parallel tsunami simulation in sam(oa)^2 are summarized in Algorithm 5.1. The simulation consists of three phases: *initialization*, *computation* and *finalization*. In the initialization phase, the application first creates a grid and its cell-associated data structures for $\vec{\mathbf{q}} = (\mathbf{q}_1, \ldots, \mathbf{q}_n)$, where $\mathbf{q}_j = (h, hu, hv, b)$. Then it refines the initial grid and interpolates $\vec{\mathbf{q}}$ until the user-defined maximal refinement level is reached. Following this, it reads the initial displacement data of the ocean floor b and water height h as resulting from an earthquake simulation. hu and hv are set to 0 as the ocean is assumed initially at rest.

The tsunami simulation is achieved in the computation phase with a while loop starting at time $t = 0$. At iteration of simulation time t, the application first refines and coarsens the grid and performs interpolation and restriction on $\vec{\mathbf{q}}^{(t)}$ accordingly. It also balances loads by redistributing and repartitioning sections. As mentioned in Section 5.2.3, grid refinement and load balancing can occur in different orders depending on the parallelization approach chosen. The program then computes the next time step Δt, which must globally satisfy a stability condition involving the wave propagation velocity. The following computation of the new cell states $\vec{\mathbf{q}}^{(t+\Delta t)}$ involves communication, i.e., exchange of boundary cell layers between neighboring processes. A flag for refinement or coarsening is then set for each cell. The grid is controlled by user-defined minimal and maximal refinement levels to stay within a reasonable range such that it captures necessary details yet is feasible for computation. All these operations are incorporated into grid traversals as the program is operating on streams and stacks only, and no random or index-based access to individual cells is allowed. At the end of the iteration, visualization output is produced if the condition is met.

Algorithm 5.1: Main algorithm of the parallel tsunami simulation from sam(oa)^2

1 **Function** `Main()`**:**
2   `MPI_Init()`
3   Initialize numRanks, rank, etc.
    // 1. Initialization
4   Traversal: Create grid, initialize $\vec{\mathbf{q}}^{(t=0)}$, set refinement flags
5   **while** *refinement flags are set* **do**
6   Traversal: Refine grid, interpolate $\vec{\mathbf{q}}$, balance load
7   Traversal: Initialize $\vec{\mathbf{q}}$, set refinement flags
8   **end**
9   Traversal: Load bathymetric data b and initial water height h
10     $t \leftarrow 0$
    // 2. Computation: tsunami simulation main loop
11   **while** $t < t_{\max}$ **do**
12   Traversal: Refine grid, interpolate/restrict $\vec{\mathbf{q}}^{(t)}$, balance load
13   Traversal: Compute time step Δt
14   Traversal: Compute $\vec{\mathbf{q}}^{(t+\Delta t)}$ with communication, set refinement flags
15         $t \leftarrow t + \Delta t$
16   **if** *every K iterations* **then**
17   Write visualization output $\vec{\mathbf{q}}^{(t)}$
18   **end**
19   **end**
    // 3. Finalization
20   `MPI_Finalize()`
21 **End**

5.3 Resource-elastic Transformation

In this section, we discuss how to transform the parallel tsunami simulation shown in Algorithm 5.1 into a resource-elastic application with the programming models discussed in Chapter 4. Based on the centralized design of the Elastic MPI framework, malleable applications must periodically react upon the resource change decisions made by the resource manager. Therefore, they should be designed in such a way that they can run on any amount of resources with the ability to expand and reduce on resources at any time.

Based on the performance analysis of the tsunami simulation presented in [77], we know that in Algorithm 5.1, the tsunami simulation while loop dominates the computation. The initial grid refinement loop in the initialization phase normally takes a few iterations that contribute to only a small fraction of the total execution time. Therefore, resource adaptivity should be restricted to the simulation loop only. Observing that the program is communication-intensive and contains a single computation loop, it can be modified using Algorithm 4.2 as a guideline. The major steps of the tsunami simulation after resource-elastic transformation are summarized in Algorithm 5.2 and 5.3.

Main function: preexisting processes

One crucial step in Elastic MPI programming is the separation of the preexisting and `JOINING` processes, such that they execute their respective part of a program and merge at a point where the computation can be carried on without interruption. The original processes being created during program start must execute the initialization steps, i.e., the initial domain decomposition and distribution. Algorithm 5.2 line 4 to 14 reflect this step.

Entering the main simulation loop, the preexisting processes perform the same steps as in the original program (Algorithm 5.1) except for an additional if-block for resource adaptation at the end of the iteration.

The conditional controls the frequency of probing the resource manager. As discussed in Section 4.2, probing should not occur too often or too infrequently, and it is ideal to overlap data redistribution from resource adaptation with the application's inherent load balancing step. For this application, since load balancing already takes place at each iteration (line 16), resource adaptation can be performed as frequently as every iteration in that regard.

Main function: routing of the `JOINING` processes

The `JOINING` processes, after being initialized with some essential variables, are redirected to the resource adaptation block immediately bypassing the *initialization* phase, because firstly, they would receive proper data-preparation in the resource adaptation block, and secondly, the *initialization* phase of this application is relative intensive thereby should be avoided.

Upon returning from the adaptation block, the `JOINING` processes continue from line 15 and join in the computation at a new iteration. At the same time, the preexisting processes return from their resource adaptation entry point (line 26) and also start from line 15 for a new iteration. At this point, the two process groups successfully merge together.

In the computation loop, the resource adaptation block entry can be placed at either the beginning or the end of the iteration without interfering with the computation. However, placing it at the beginning is less optimal, because when the `JOINING` processes enter the

Algorithm 5.2: Main algorithm of the Elastic MPI tsunami simulation

```
1  Function Main():
2      MPI_Init_adapt()                          // returns process status in procStatus
3      Initialize numRanks, rank, etc.
       // 1. Initialization: JOINING processes bypass
4      if procStatus = JOINING then
5          Resource_Adapt()                      // loop counter t is synced
6      else
7          Traversal: Create grid, initialize q⃗^(t=0), set refinement flags
8          while refinement flags are set do
9              Traversal: Refine grid, interpolate q⃗, balance load
10             Traversal: Initialize q⃗, set refinement flags
11         end
12         Traversal: Load bathymetric data b and initial water height h
13         t ← 0
14     end
       // 2. Computation: tsunami simulation main loop
15     while t < t_max do
16         Traversal: Refine grid, interpolate/restrict q⃗^(t), balance load
17         Traversal: Compute time step Δt
18         Traversal: Compute q⃗^(t+Δt) with communication, set refinement flags
19         t ← t + Δt
20         if every K iterations then
21             Write visualization output q⃗^(t)
22         end
23         if Time to probe Resource Manager then
24             MPI_Probe_adapt()                 // returns adapt decision in adaptFlag
25             if adaptFlag says to adapt then
26                 Resource_Adapt()
27             end
28         end
29     end
       // 3. Finalization
30     MPI_Finalize()
31 End
```

Algorithm 5.3: Resource adaptation function of the Elastic MPI tsunami simulation

```
32 Function Resource_Adapt():
33     MPI_Comm_adapt_begin()
34     if there are LEAVING processes then
           // No need to load balance as it occurs at next iteration beginning
35         LEAVING processes send their sections to STAYING processes
36     end
37     if there are JOINING processes then
38         JOINING processes create and initialize grid and data objects
39         Sync data (loop counters t, etc.)
40     end
       // Data redistribution is omitted as it occurs at next iteration beginning
41     MPI_Comm_adapt_commit()              // MPI_COMM_WORLD is updated
42     Update numRanks, rank, procStatus, etc.
43 End
```

loop, instead of start computing directly, they check for resource adaptation again which is unnecessary.

Resource Adaptation

The `Resource_Adapt` function in Algorithm 5.3 opens the adaptation window. When there are `LEAVING` processes, they send their sections to some of the `STAYING` processes. At this step, the `STAYING` processes merely serve as temporary data holders. Load balancing is not taken care of at this point, because a load balancing step will occur at the beginning of the next iteration (line 16 in Algorithm 5.2).

For the case of resource expansion, the `JOINING` processes are initialized with data containers and objects. The loop counter (simulation time t) is synchronized between the preexisting and `JOINING` processes. Comparing to the resource adaptation block in Algorithm 4.2, the data redistribution step is omitted, the reason is also ascribed to line 16 in Algorithm 5.2.

Overall, due to the application's inherent load balancing scheme, data migration activities inside the adaptation window are reduced to a minimum, thus so is the resource adaptation overhead.

5.4 Performance Evaluation

In this section, we present the performance analysis on several test cases of a chosen benchmark simulation scenario – the Tohoku tsunami from 2011 that was preceded by a magnitude nine earthquake near the coast of Japan. For the bathymetric data of the northern Pacific ocean and the Sea of Japan, the GEBCO 2014 Grid version 20150318 is used [100]. We first assess the impact of runtime resource adaptivity on the application, i.e., the overhead introduced by resource adaptation. We then compare execution time and resource efficiency of the Elastic MPI implementation with the static MPI counterpart.

5.4.1 Execution Environment: SuperMUC

All tests for this application were conducted on the SuperMUC Petascale System [75] at the LRZ located in Garching, Germany. They were run on the supercomputer's Phase 1 thin nodes, which consists of Intel Sandy Bridge-EP Xeon E5-2680 cores, with 16 CPUs per node operating at 2.7 GHz. Table 5.1 enumerates specification details of some CPU clusters in SuperMUC. We ran our tests with 16 MPI processes per node without hyperthreading, pinning each process to a core.

All tests were pure MPI runs without the use of OpenMP. This is because in the current Elastic MPI implementation, resource adaptivity are purely MPI-based, and we want direct analysis and comparison without distraction from other factors such as threading.

The SuperMUC uses IBM Load Leveler for resource management. Due to a lack of administrative rights, we could not replace the system resource manager with the Elastic MPI resource manager. For our experimentation, we created batch scripts that allowed for running our customized resource manager nested in a batch job.

The Elastic MPI infrastructure currently supports node-level resource allocation and adaptation only (see Section 3.4). A node (16 MPI processes) is the minimal resource unit an elastic application can acquire or remove. Nodes cannot be partially utilized by an application nor shared among multiple applications. The master node (the node containing MPI Rank 0) of an application is persistent, i.e., it remains unchanged in the resource allocation throughout execution.

Table 5.1: The SuperMUC CPU cluster specifications

CPU Cluster	Phase I Fat Nodes	Phase I Thin Nodes	Phase II Haswell Nodes
Processor	Intel Westmere Xeon E7-4870 10C	Intel Sandy Bridge Xeon E5-2680 8C	Intel Haswell Xeon E5-2697 v3
Nominal frequency [GHz]	2.4	2.7	2.6
Cores per node	40	16	28
Memory per core [GByte]	6.4	2	2.3
Memory per node [GByte]	256	32	64
Memory bandwidth per node [GB/s]	136	102	137
Number of nodes	205	9216	3072
Double prec. flops per node [Gflops]	384	346	1165
Double prec. flops total [Tflops]	79	3190	3580
Interconnect	Infiniband QDR	Infiniband FDR10	Infiniband FDR14

5.4.2 The Benchmark: Simulation of the Tohoku Tsunami

Figure 5.4 displays the visualization of the tsunami wave (left), the corresponding adaptive grid (right top) as well as the resource utilization (right bottom) of the benchmark simulation of the Tohoku tsunami at two different time steps: one at simulated time one hour, the other at simulated time three hours.

The simulation was run in a 32-node elastic environment, started with 1 node (16 MPI processes) initially. The color blocks in the grid correspond to the color of the executing nodes, e.g., the blue portion of the grid is computed by the node in the same color. Nodes in gray color mean they are not utilized. At simulated time one hour, the simulated was

executed on 16 nodes with 256 processes. At three hours, it was expanded to 28 nodes with 448 processes.

The simulation grids are very refined in the area between the coast of Japan and the tsunami wave front, and very coarse for the rest of the ocean. Comparing with the grid in Figure 5.4(a), the grid in Figure 5.4(b) contains much more grid cells as the tsunami wave front propagates further and the refined region is much larger. Due to the increase in computational workload, the simulation utilizes more nodes at simulated time of three hours.

5.4.3 Resource Adaptation Overhead

One major concern of introducing runtime resource adaptivity is that the induced overhead could be significant. Not only the additional Elastic MPI function calls introduce overhead, but also do the necessary communication due to resource changes, e.g., moving data from `LEAVING` processes, synchronization between preexisting and new resources, load rebalancing operations, etc.

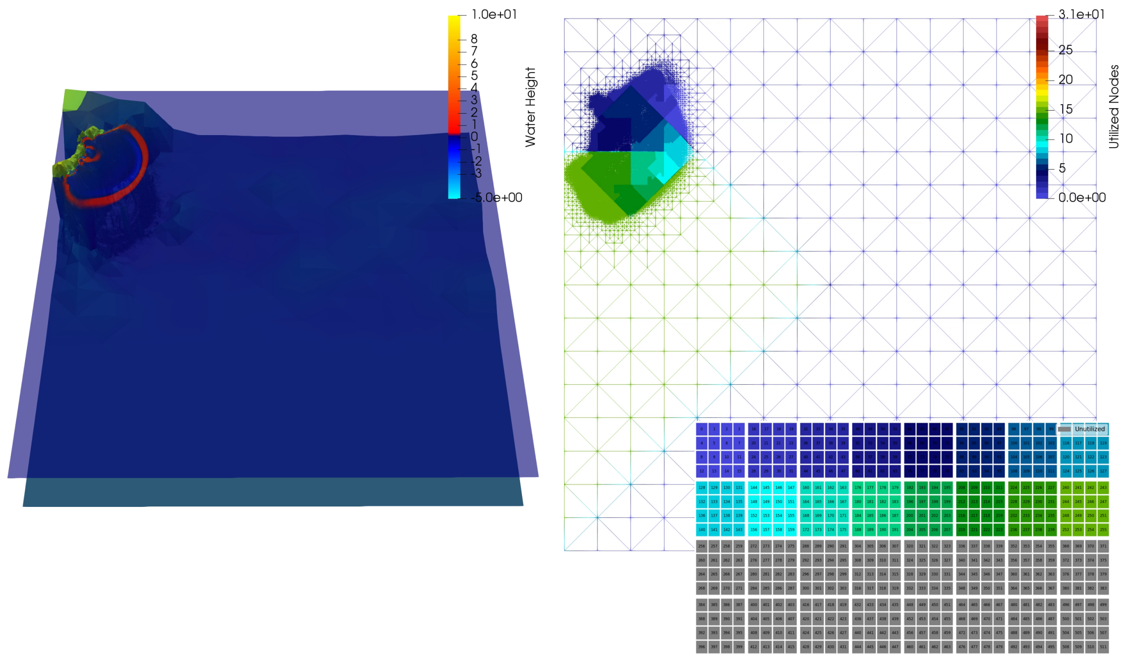
In this experiment, we want to analyze the impact of resource adaptation on the tsunami simulation. In each test, we run a single instance of the benchmark in a 32-node elastic environment. It always starts with minimal resources: 1 node (16 processes). The maximal resources it can expand to is 32 nodes (512 processes).

In order to fully examine different cases of resource expansion and reduction, we use a random elastic runtime scheduler that gives new resource assignment between 1 and 32 nodes every 60 seconds. There is no adaptation if the generated random number is the same as the current number of nodes. The application is set to probe the resource manager every 50 iterations, which is approximately every few seconds.

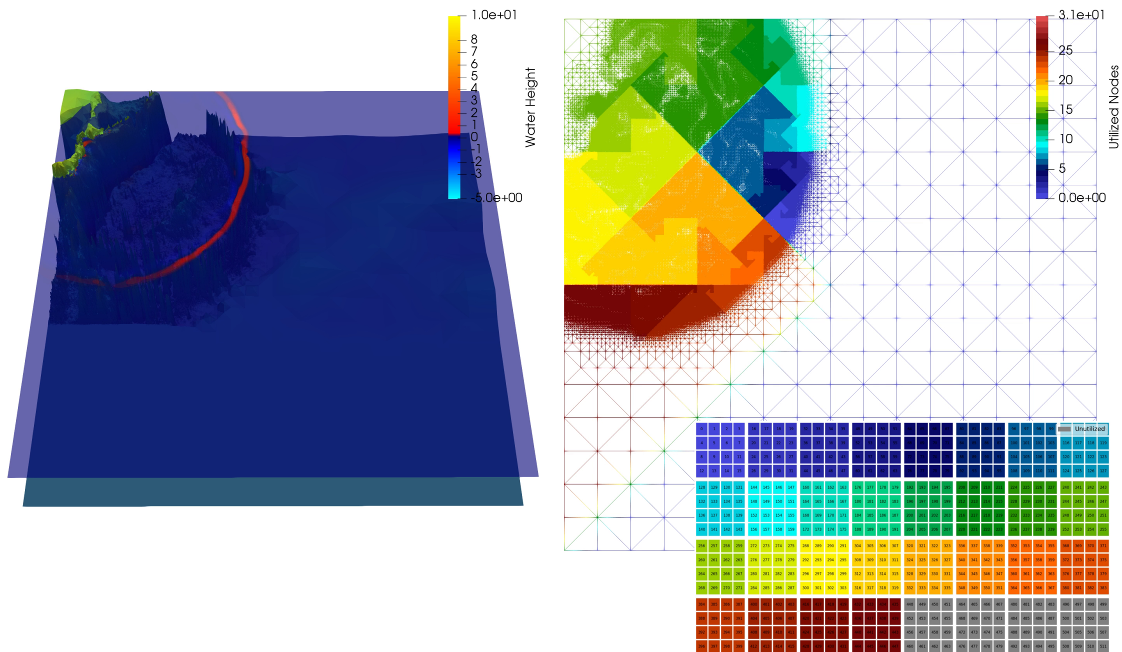
As an example, the resource change profile of one of the test runs is selected and shown in figure 5.5. In this run, resource adaptation took place 4 times: it first expanded from 1 node (16 processes) to 18 nodes (288 processes), then again to 24 nodes (384 processes), afterwards, it reduced to 14 nodes (224 processes), and later again to 2 nodes (32 processes).

Table 5.2 shows the average execution time of the Elastic MPI functions. These measurements vary depending on the type and size of the adaptation. As a general rule of thumb, execution times of `MPI_Comm_adapt_begin` and `MPI_Comm_adapt_commit` are longer for bigger resource changes, e.g., expanding from 1 to 20 nodes is a bigger change than expanding from 18 to 20 nodes, And these functions also take longer for resource expansion than reduction. An in-depth performance analysis on the Elastic MPI functions can be found in [65, 12]. In each test, `MPI_Init_adapt` is called only once at the beginning, while `MPI_Probe_adapt` is called every 50 iterations. `MPI_Comm_adapt_begin` and `MPI_Comm_adapt_commit` are called as many times as the number of adaptations, which is on average 4 to 5 times per test.

The execution time of the simulation is broken down into 6 major computational tasks: grid refinement, grid conformity check, load balancing, time step computation, resource adaptation, and other operations such as I/O. For the test run shown by Figure 5.5, its execution time percentage spent on each task is listed in Table 5.3. Resource adaptation overhead is 5.7% of the total execution time. This includes the execution time of the Elastic MPI functions as well as the time measured around all adaptation windows. From all tests with a random elastic scheduler, their resource adaptation overhead is normally measured between 5% and 10% of the total execution time.



(a) Simulation time at one hour, executing on 16 nodes (256 MPI processes)



(b) Simulation time at three hours, executing on 28 nodes (448 MPI processes)

Figure 5.4: Two time steps of the benchmark simulation of the Tohoku tsunami from 2011 that was preceded by a magnitude 9 earthquake near the coast of Japan. The tsunami simulation is visualized on the left, and the underlying adaptive grid as well as real-time resource utilization is displayed on the right. The color blocks in the grid correspond to the color of the executing nodes, e.g., the blue portion of the grid is computed by by the node in the same color. Nodes in gray color are not utilized.

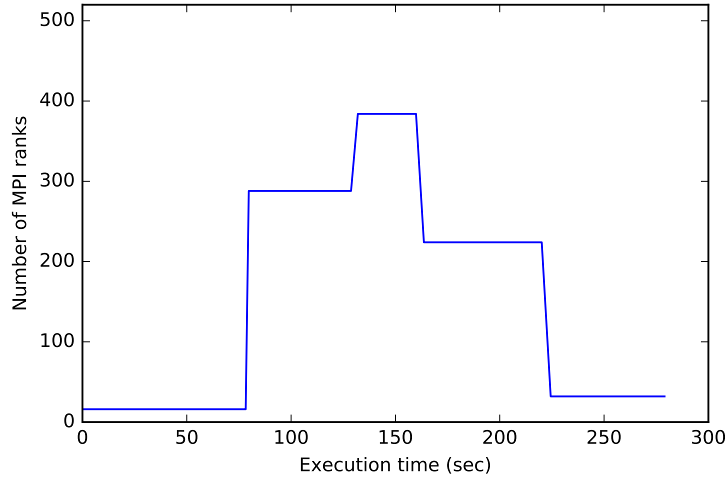


Figure 5.5: Resource change profile during a test run. Minimum resources is 1 node (16 processes), maximum resources is 32 nodes (512 processes). Four resource adaptations took place during this test run: firstly an expansion from 1 to 18 nodes (from 16 to 288 processes), secondly an expansion from 18 to 24 nodes (from 288 to 384 processes), thirdly a reduction from 24 to 14 nodes (from 384 to 224 processes), and lastly a reduction from 14 to 2 nodes (from 224 to 32 processes).

Table 5.2: Average execution time of elastic MPI functions

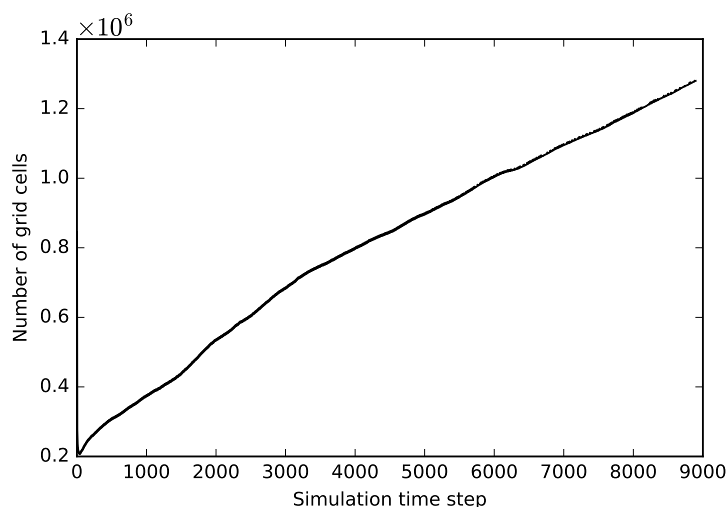
Elastic MPI function	**Avg. exec. time (sec)**
`MPI_Init_adapt`	0.04
`MPI_Probe_adapt`	0.06
`MPI_Comm_adapt_begin`	2.78
`MPI_Comm_adapt_commit`	0.10

Table 5.3: Average exec. time percentage of computational tasks

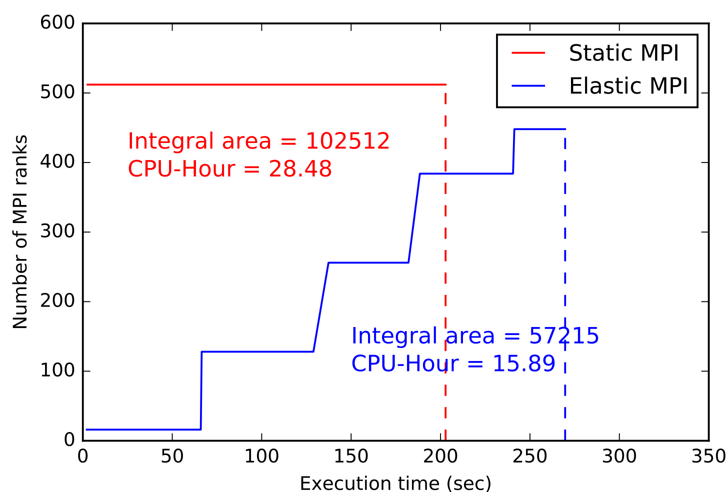
Computational task	**% total exec. time**
Grid refinement	24.6%
Grid conformity check	13.5%
Load balancing	22.1%
Time step computation	30.5%
Resource adaptation	5.7%
Others	3.6%

Table 5.4: Comparison of execution time and CPU hours

Test run	Exec. time (sec)	CPU hours
Static MPI with 32 nodes (512 processes)	203	28.48
Elastic MPI with stepping scheduler	270	15.89



(a) Number of grid cells in tsunami simulation phase



(b) Number of MPI processes vs. execution time.

Figure 5.6: Dynamic workload behavior of the Tohoku benchmark. And the performance comparison between an Elastic MPI run and a static MPI run on a 32-node environment,

5.4.4 Runtime and Resource Efficiency

In this experiment, we want to analyze the application's execution time and resource efficiency by comparing static and Elastic MPI test runs in identical execution environments. Because the performance monitoring functionality and the decision logic in the resource manager are not fully implemented, our Elastic MPI tests are conducted with predefined runtime schedulers.

The number of grid cells is the best indicator for computational workload. To understand the dynamic behavior of the application, we first obtain a growth profile of the grid cell count from the simulation phase as shown by Figure 5.6(a). One can observe an almost linear growth from 0.2 to 1.28 million cells in about 9000 simulation time steps. This

indicates an increase in workload by more than 6 times from the beginning to the end, which is a drastic change.

Note that the number of grid cells will not grow infinitely. It will eventually saturate and decrease when the simulation runs long enough such that the wave front propagates out of the simulation domain. But in the current test scenario, the simulation would not reach the cell number saturation or decrease stage.

In the perfect scenario, the resource manager should be able to capture such a workload behavior and assign more resources to the application accordingly. In order to simulate this scenario, we use a stepping elastic scheduler that increases the resource assignment every 60 seconds.

For fair comparison, the execution environment for both runs are kept the same: 32-nodes with 16 processes per node. From performance analysis done in [77], we know that the application can scale on more than 32-nodes. Therefore, we try to utilize all available resources in the static MPI run, which is 32 nodes (512 processes). For the elastic run, the application starts will 1 nodes (16 processes) and expands to more nodes according to the stepping scheduler.

Figure 5.6(b) shows the number of MPI processes versus execution time for the two runs. Table 5.4 further lists their execution time and CPU hours. The total execution time is 203 seconds for the static run, and 270 seconds for the elastic run. The CPU hours can be computed by integrating the number of MPI processes (each process is pinned to a CPU) over the execution time, which makes 28.48 CPU hours for the static run and 15.89 CPU hours for the elastic run. Even though the execution time of the Elastic MPI run is about $\frac{1}{3}$ longer than that of the static run, its CPU hours are reduced almost by $\frac{1}{2}$, meaning that it is almost twice as resource efficient as the static run.

5.5 Summary

As the first case study for malleable software development with Elastic MPI, we selected a tsunami simulation from the sam(oa)^2 framework. It represents the classical communication-intensive grid-based HPC applications implementing the SPMD model. This application is compute-bound and has a very dynamic computational workload.

For communication-intensive applications, data redistribution and load balancing are expensive operations. This poses challenges in realizing resource elasticity, because frequent resource changes (which require frequent data redistribution) can introduce significant overhead. The selected tsunami simulation, however, has a built-in load balancing scheme to mitigate load imbalances caused by AMR. Our resource-elastic implementation overlaps these built-in operations with those required by resource changes, thereby minimizes the overhead. This finding is backed by the results from tests with a random elastic scheduler. The resource adaptation overhead was typically measured between 5% and 10% of the total execution time.

The tsunami simulation demonstrated a very dynamic behavior in its computational workload. When runtime resource allocation was adjusted in accordance with its workload, the Elastic MPI test run showed a significant reduction in CPU hours compared to a static MPI test run in the same execution environment. From this we can conclude that for applications with dynamic workload behavior, resource elasticity can help to improve resource efficiency on the application level.

The largest Elastic MPI tests on scalability we have conducted (not presented in this dissertation) were on 128 thin nodes and 64 Haswell nodes on SuperMUC. We were not able to scale further due to some known stability issues in the current Elastic MPI release (discussed in Section 3.4.5). For these reasons, we were only able to obtain completed test runs of the tsunami simulation on 32 thin nodes. Tests in larger environments with more than 32 nodes have failed. In those tests, we were able to launch the application and had it run and adapt for a period time, but the application crashed before completion. Were these stability issues in the infrastructure fixed, we should be able to run tests on larger scales.

6 Elastic Parallel Oil Reservoir Simulation with 2.5-D Adaptive Mesh

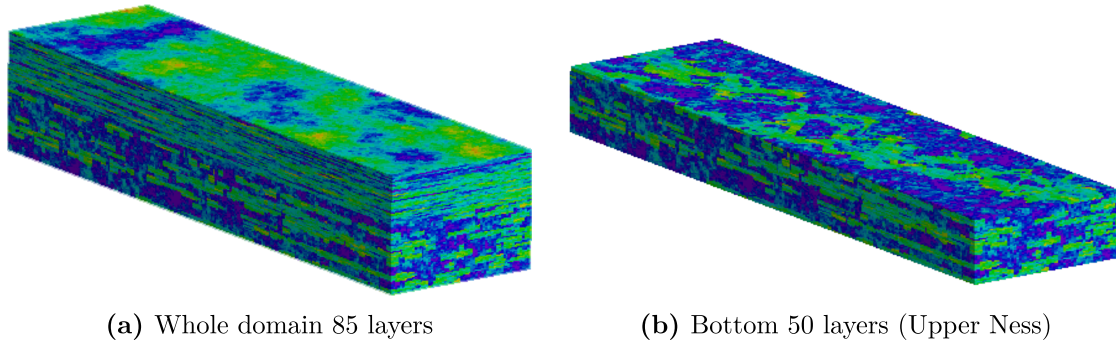
HPC applications rely on efficient utilization of computational resources and the system's communication network. Their performance is often limited by either the processor speed, memory or communication bandwidth. Applications are considered *compute-bound* (or *CPU-bound*) when their progress is limited by the processor speed. This types of application often have high processor utilization rates and are easier to scale. Applications are considered *communication-bound* (or *bandwidth-bound*) when their progress is limited by the networking due to intensive communication among the executing processes. These applications push the limits of the network bandwidth and are sensitive to communication overhead, thus harder to scale.

The oil reservoir simulation is yet another classical communication-intensive grid-based HPC application implementing a SPMD execution model. Similar to the tsunami simulation presented in Chapter 5, it is also adopted from sam(oa)^2 [77] and has a similar computational workflow, because it is likewise developed upon adaptive meshes based on Sierpiński traversal. As a second application selected for Elastic MPI investigation, however, it also possesses different characteristics:

1. Based on very different underlying physics, it has a much more intensive computation kernel compared to the tsunami simulation. It is a 3-D simulation implemented with 2.5-D adaptive meshes, i.e., 3-D meshes with adaptivity in two dimensions.
2. It has a mostly constant computational workload.
3. Last but not least, while the tsunami simulation is compute-bound, this simulation demonstrates more communication-bound characteristics, which means it is more sensitive to communication overhead and harder to scale.

We are interested in experimenting with applications with different parallel performance characteristics, i.e., with both compute-bound and communication-bound applications, because resource adaptivity has major impacts on communication overhead and these applications have very different sensitivity levels to that. Theoretically speaking, the performance of communication-bound applications is more impaired by resource adaptivity, so the improvement on resource efficiency is expected to be less than that of compute-bound applications. Being communication-bound and having a constant workload, the oil reservoir simulation is a good candidate to be included as it adds variety to our application base.

In this chapter, we first explain the target simulation scenario along with its domain discretization treatment, i.e., the 2.5-D adaptive mesh. Then, we briefly discuss the physics used to model the simulation and its implementation in sam(oa)^2. Following that, we apply the resource-elastic transformation using the Elastic MPI library. Lastly, we present the application-level performance and resource efficiency analysis conducted from an isolated environment with single application.



(a) Whole domain 85 layers **(b)** Bottom 50 layers (Upper Ness)

Figure 6.1: Permeability field of the SPE10 benchmark simulation domain, with a resolution of $220 \times 60 \times 85$ grid cells. The top part, consisting of 35 layers, is a *Tarbert* formation that is a representation of a prograding near shore environment. The lower part, consisting of 50 layers, is a fluvial *Upper Ness* formation. Image source [101].

6.1 The SPE10 Benchmark Simulation Scenario

This application simulates a scenario defined by the Society of Petroleum Engineer's tenth benchmark problem (SPE10) first published in 2000 [101], which describes oil production by water injection in a cuboid domain. The domain of the benchmark is filled with soil, a porous medium that permits the flow of liquids. The lateral extent of the domain is on the order of kilometers, corresponding to the *field scale*, and pores are regarded as too small to be resolved. Permeability is highly heterogeneous throughout the domain, and so is porosity. Figure 6.1 shows the permeability field with a resolution of $220 \times 60 \times 85$ grid cells. The whole domain consists of two distinct layers of formation: the *Tarbert* formation (top 35 layers), which is a representation of a prograding near shore environment; and the fluvial *Upper Ness* formation (bottom 50 layers).

The domain is initially saturated with oil. A vertical well located at the center of the domain injects water at a constant rate. Four vertical production wells at the corners extract oil by pressure-induced flow. Over time, the reservoir fills with water that spreads out from the center and displaces the oil towards the production wells at the corners. Figure 6.2 shows the saturation profile in of a full size SPE10 benchmark simulation with 85 vertical layers. Water spreads from the center injection well to the corner production wells over time. Due to the heterogeneity in material permeability, water propagation speed differs across the different vertical layers.

2.5-D Adaptive Mesh Based on Sierpiński Traversal

Unlike some other SFCs such as Hilbert curves, which can be applied on higher dimensional domains, the Sierpiński curve is limited to 2-D domains. To extend a 2-D grid based on

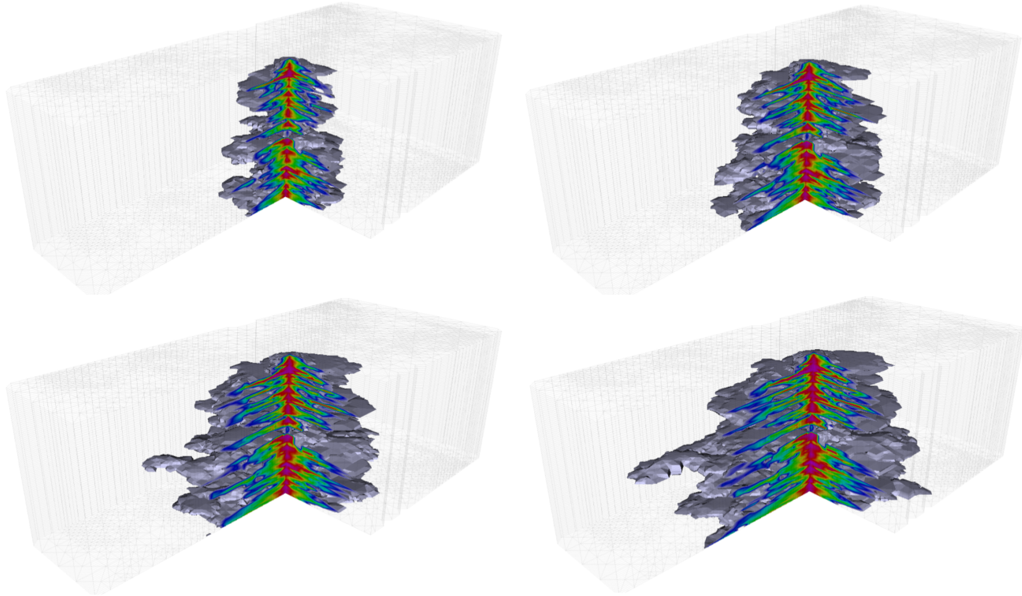
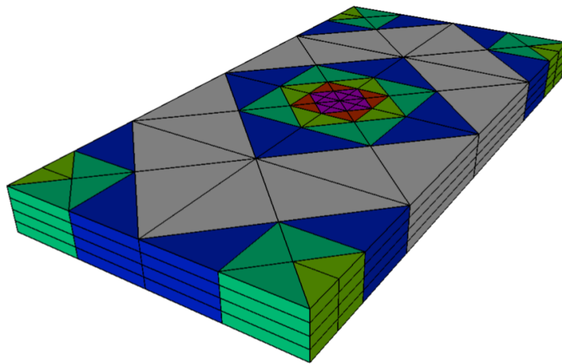
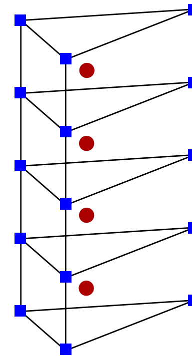


Figure 6.2: Full-sized SPE10 simulation with 85 vertical layers. Saturation profile at 25, 50, 75 and 100 simulated days. One corner of the domain has been clipped for better visibility. Image source [77].



(a) Prismatic grid with 4 layers in sam(oa)^2. Image source [77]



(b) A vertical array of prism cells

Figure 6.3: A 2.5-D adaptive prismatic grid with four layers in sam(oa)^2. In each 2-D position (x, y), an array of four cell data is stored, with each data associated with an additional z-coordinate. The vertex data has the same z-major storage scheme, with one more element in each vertical array than the cell data.

Sierpiński traversal into a 3-D domain, layers of the 2-D grid are stacked on top of one another to create depth in the vertical dimension. The resulting grid is the so-called 2.5-D adaptive mesh with a fixed number of layers, i.e., the grid may be dynamically refined and coarsened in the horizontal dimensions, but the number of vertical layers is constant. Grid cells in the 2.5-D adaptive mesh take the shape of a prism instead of a triangle. Figure 6.3 shows a 2.5-D adaptive mesh with four layers and a vertical array of prism cells.

Due to the constant size in the vertical dimension, an extension to the 2-D grid storage scheme is implemented by storing a fixed size array at each 2-D position instead of a single element. This is applicable to both vertex and cell data. Figure 6.3(b) shows a vertical array of prism cells at a certain position (x, y) of the 4-layer grid. Cell data (represented by a red dot) is located at the center of each cell. Instead of a single element, the cell data stored at (x, y) contains an array of 4 elements, and each of which is associated with a third coordinate z. The same procedure is applied to vertex data. The only difference is that a vertical vertex data array (represented by blue squares) has one more element than a cell data array. This z-major storage scheme allows for kernel vectorization over the z-dimension when the number of layers is large enough. More details on the implementation of the 2.5-D adaptive mesh can be found in [102, 77].

6.2 Porous Media Flow Simulation in sam(oa)^2

In this section, we discuss the modeling of the porous media flow and the implementation of the simulation in sam(oa)^2. Content of the rest of this section is a short recapitulation based on [77].

6.2.1 Modeling of Immiscible Porous Media Flows

We consider a domain filled with permeable materials such as soil, sand or grain that permits flow of liquids or gases, which are mixtures of two immiscible substances, such as water and oil in oil reservoirs or water and gas in carbon reservoirs. The substances in the liquid or gas mixture are called *liquid phases*, denoted by $\alpha \in \{w, n\}$, where w represents water, the *wetting phase* and n represents oil (or gas), the *non-wetting phase*.

The domain is usually a whole or partial reservoir. Due to its large lateral extent, usually on the order of kilometers, the pores in the medium are considered too small to be resolved, and so are the liquid interfaces. A space- and time-dependent averaged quantity $s_\alpha(x, y, z, t)$ is introduced to describe the saturation of phases, i.e., the fraction per unit volume. Saturation is a relative quantity, therefore, for two-phased liquids, there is

$$s_w + s_n = 1. \tag{6.1}$$

Considering the fact that phases are never fully saturated nor desaturated, concepts of *effective saturation* denoted by $s_{\alpha e}$ and *residual saturation* denoted by $s_{\alpha r}$ are introduced. In computation, only the effective saturation of phases, given by

$$\begin{aligned} s_{we} &:= \frac{s_w - s_{wr}}{1 - s_{wr} - s_{nr}} \\ s_{ne} &:= \frac{s_n - s_{nr}}{1 - s_{wr} - s_{nr}}, \end{aligned} \tag{6.2}$$

are taken into consideration, because the residual saturation is assumed to have no effect on the flow. Substituting (6.1) into (6.2) leads to

$$s_{we} + s_{ne} = 1. \tag{6.3}$$

Darcy's Law

Porous media flows are often assumed to be laminary [103], and the forces that act on the fluid are caused by the pressure gradient and gravity. Derived from the Navier-Stokes equation [104, 105] the phase velocity is given by

$$\mathbf{u}_\alpha = \lambda_\alpha(s_{\alpha e})\,\mathbf{K}(-\nabla p_\alpha + \rho_\alpha \mathbf{g}), \tag{6.4}$$

where $\mathbf{K}$ is a permeability tensor, ∇p_α is the pressure gradient, ρ_α denotes the phase density, $\mathbf{g}$ is the gravity vector, and $\lambda_\alpha(s_{\alpha e})$ represents the phase mobility, which is defined as a function of the relative permeability κ_α and the phase viscosity μ_α, given by

$$\lambda_\alpha(s_{\alpha e}) := \frac{\kappa_\alpha(s_{\alpha e})}{\mu_\alpha} \tag{6.5}$$

Without the consideration of gravity, i.e., omitting the term $\rho_\alpha \mathbf{g}$, equation (6.4) is called *Darcy's law*. With gravity, the equation is referred to as the *extended Darcy's law*.

Transport Equations

Using the extended Darcy's law (6.4), conservation of mass for each phase is expressed by the *transport equation*, given by

$$(\rho_\alpha \Phi s_\alpha)_t + \operatorname{div}(\rho_\alpha \mathbf{u}_\alpha) = \rho_\alpha q_\alpha, \tag{6.6}$$

where $\Phi(x, y, z)$ is the *porosity*, i.e., the ratio of pore volume per unit volume; $\rho_\alpha \Phi s_\alpha$ represents the mass per unit volume; $\rho_\alpha \mathbf{u}_\alpha$ denotes the flux; and $\rho_\alpha q_\alpha$ is the source term, where q_α represents the incoming/outgoing phase volume over time at a source/sink. The equation basically states that the change of mass and the transport of mass sums up to the amount of mass coming from/going into a source/sink.

Assuming the flow is incompressible, which means the phase density ρ_α is constant, equation (6.6) can be simplified as

$$(\Phi s_\alpha)_t + \operatorname{div}(\mathbf{u}_\alpha) = q_\alpha. \tag{6.7}$$

Closed Form for Incompressible Flow

The *capillary pressure* models the pressure discontinuity caused by a force induced by surface tension on phase interfaces. In our simulation problem, since the capillary pressure is small in water and oil mixtures, we neglect it and define

$$p := p_w = p_n. \tag{6.8}$$

From equation (6.3), we define

$$s := s_{we} = 1 - s_{ne}. \tag{6.9}$$

Substituting (6.8) and (6.9) into the extended Darcy's law (6.4), the phase velocities are given by

$$\begin{aligned} ad\mathbf{u}_w &= \lambda_w(s)\,\mathbf{K}(-\nabla p + \rho_w \mathbf{g}), \\ \mathbf{u}_n &= \lambda_n(1-s)\,\mathbf{K}(-\nabla p + \rho_n \mathbf{g}). \end{aligned} \tag{6.10}$$

From (6.1), we can derive $(\Phi s_n)_t = -(\Phi s_w)_t$, which can be substituted into the transport equation (6.7) and yields

$$\begin{aligned} (\Phi s_w)_t + \operatorname{div}(\mathbf{u}_w) &= q_w, \\ -(\Phi s_w)_t + \operatorname{div}(\mathbf{u}_n) &= q_n. \end{aligned} \tag{6.11}$$

Addition of the two equations returns the total volume balance

$$\operatorname{div}(\mathbf{u}_w + \mathbf{u}_n) = q_w + q_n. \tag{6.12}$$

From (6.2), s_w can be expressed with the effective and residual saturations, i.e.,

$$s_w = s_{we}(1 - s_{wr} - s_{nr}) + s_{wr}, \tag{6.13}$$

which, substituting into equation (6.11) yields

$$\begin{aligned} (\Phi(1 - s_{wr} - s_{nr})s)_t + \operatorname{div}(\mathbf{u}_w) &= q_w, \\ -(\Phi(1 - s_{wr} - s_{nr})s)_t + \operatorname{div}(\mathbf{u}_n) &= q_n. \end{aligned} \tag{6.14}$$

The term $(1 - s_{wr} - s_{nr})$ is a constant ratio and can be considered as a reduction of pore volume. Defining the modified porosity as

$$\tilde{\Phi} := (1 - s_{wr} - s_{nr})\Phi, \tag{6.15}$$

and redefining the letter Φ to describe $\tilde{\Phi}$, we obtain the simplified closed form as

$$\begin{aligned} (\Phi s)_t + \operatorname{div}(\mathbf{u}_w) &= q_w, \\ -(\Phi s)_t + \operatorname{div}(\mathbf{u}_n) &= q_n, \\ \operatorname{div}(\mathbf{u}_w + \mathbf{u}_n) &= q_w + q_n. \end{aligned} \tag{6.16}$$

The first two equations are the closed form transport equations, and the third equation describes total volume balance of the two phases.

Quasilinear Form

The *quasilinear form*, given by

$$\alpha q_t + \mathbf{f}'(q) \cdot \nabla q = \Psi(q), \tag{6.17}$$

resembles the advection equation with a flux derivative $f'(q)$ and a source term $\Psi(q)$. For numerical analysis, it is useful to transform (6.16) into the quasilinear form, because the matrix $f'(q)$ contains the signal speeds of the system [90], which can be used for computation of the time steps.

The first step for obtaining the quasilinear form is to substitute (6.8) and (6.9) into the extended Darcy's law (6.4) and obtain the expression for each phase velocity, i.e.,

$$\begin{aligned} \mathbf{u}_w &= \lambda_w(s)\mathbf{K}(-\nabla p) + \rho_w\lambda_w(s)\mathbf{K}\mathbf{g}, \\ \mathbf{u}_n &= \lambda_n(1-s)\mathbf{K}(-\nabla p) + \rho_n\lambda_n(1-s)\mathbf{K}\mathbf{g}, \\ \Rightarrow \quad \mathbf{u}_T &= \left[\lambda_w(s) + \lambda_n(1-s)\right]\mathbf{K}(-\nabla p) + \left[\lambda_w(s)\rho_w + \lambda_n(1-s)\rho_n\right]\mathbf{K}\mathbf{g}, \end{aligned} \tag{6.18}$$

where $\mathbf{u}_T := \mathbf{u}_w + \mathbf{u}_n$. By rearranging terms, the phase velocities can be expressed as

$$\begin{aligned} \mathbf{u}_w &= \frac{\lambda_w(s)}{\lambda_w(s) + \lambda_n(1-s)}(\, \mathbf{u}_T + \lambda_n(1-s)(\rho_w - \rho_n)\mathbf{K}\mathbf{g}\,), \\ \mathbf{u}_n &= \frac{\lambda_n(1-s)}{\lambda_w(s) + \lambda_n(1-s)}(\, \mathbf{u}_T + \lambda_w(s)(\rho_n - \rho_w)\mathbf{K}\mathbf{g}\,). \end{aligned} \tag{6.19}$$

Inserting (6.19) into (6.16) and applying the chain rule to the resulting equations with some rearrangement, the quasilinear form for each phase is obtained. For the wetting phase, it is given by

$$\Phi s_t + \xi_w \cdot \nabla s = \Psi(s), \tag{6.20}$$

where

$$\begin{aligned} \xi_w &:= \frac{\lambda_w'(s)\lambda_n(1-s) - \lambda_w(s)\lambda_n'(s)}{(\lambda_w(s) + \lambda_n(1-s))^2}\mathbf{u}_T + \frac{\lambda_w'(s)\lambda_n^2(s) + \lambda_w^2(s)\lambda_n'(s)}{(\lambda_w(s) + \lambda_n(1-s))^2}(\rho_w - \rho_n)\mathbf{K}\mathbf{g}, \\ \Psi(s) &:= q_w - \frac{\lambda_w(s)}{\lambda_w(s) + \lambda_n(1-s)}(q_w + q_n) - \frac{\lambda_w(s)\lambda_n(1-s)}{\lambda_w(s) + \lambda_n(1-s)}(\rho_w - \rho_n)\,\mathrm{div}(\mathbf{K}\mathbf{g}). \end{aligned} \tag{6.21}$$

The same equation is obtained for the non-wetting phase with $\xi_n = \xi_w$.

6.2.2 Discretization and Numerical Solution

The PDE system (6.16) can be solved implicitly or semi-implicitly with mixed discretization schemes [106, 107, 108]. sam(oa)^2 implements the IMPES (IMplicit Pressure, Explicit Saturation) scheme that alternates between implicit solver for the pressure and explicit solver for the saturation [109]. The IMPES scheme is easier to compute than a fully implicit formulation and is the canonical choice for the SPE10 benchmark [107].

Discretizing the first transport equation in (6.16) with the cell-centered finite volume method, an explicit update rule for the saturation of each cell j is obtained as

$$s_j^{(t+\Delta t)} = s_j^{(t)} + \frac{\Delta t}{\Phi_j V_j}\left((Q_w)_j - \sum_{i\in\mathcal{N}(j)} A_{j,i}\mathcal{F}_w(s_j^{(t)}, s_i^{(t)})\right), \tag{6.22}$$

where Δt denotes the time step, V_j is the cell volume and $\Phi_j V_j$ represents the effective cell volume, $(Q_w)_j$ denotes the discrete source term that is nonzero only near wells, $\mathcal{N}(j)$ represents the set of all neighboring cells of cell j, $A_{j,i}$ is the area of the intersecting surface between cell j and its neighbor cell i, and $\mathcal{F}(\dots)$ represents the net update between the adjacent cell pair.

Deriving from the discrete transport equation (6.22) for both phases, an equation describing the total volume balance is obtained as

$$\sum_{i\in\mathcal{N}(j)} A_{j,i}\Big(\mathcal{F}_w\left(s_j^{(t)}, s_i^{(t)}\right) + \mathcal{F}_n\left(s_j^{(t)}, s_i^{(t)}\right)\Big) = (Q_w)_j + (Q_n)_j. \tag{6.23}$$

Applying a finite element method to discretize pressure p, we obtain for (6.23) a non-linear system that must be solved in each IMPES time step.

Solving for Pressure and Upstream Mobility

$\mathcal{F}_w(s_l, s_r)$ and $\mathcal{F}_n(s_l, s_r)$ are numerical flux solvers that compute the net updates of saturation on an interface between a left cell l and a right cell r. In these solvers, sam(oa)^2 employs an *upstream differencing* scheme for an approximation to the Riemann solution. Defining

$$\mathbf{u}_T := \mathcal{F}_w(s_l, s_r) + \mathcal{F}_n(s_l, s_r), \tag{6.24}$$

we solve (6.23) for $\mathbf{u}_T$.

In order to close the system, we define the multidimensional upstream formula given by

$$\begin{aligned} \mathcal{F}_w(s_l, s_r) &:= \lambda_w^* \, \mathbf{n}_{j,i}^T \mathbf{K}(-\nabla p + \rho_w \mathbf{g}) \\ \mathcal{F}_n(s_l, s_r) &:= \lambda_n^* \, \mathbf{n}_{j,i}^T \mathbf{K}(-\nabla p + \rho_n \mathbf{g}), \end{aligned} \tag{6.25}$$

where

$$\begin{aligned} \lambda_w^* &:= \begin{cases} \lambda_w(s_l), & \text{if } \mathbf{n}_{j,i}^T \mathbf{K}(-\nabla p + \rho_w \mathbf{g}) > 0. \\ \lambda_w(s_r), & \text{otherwise.} \end{cases} \\ \lambda_n^* &:= \begin{cases} \lambda_n(1 - s_l), & \text{if } \mathbf{n}_{j,i}^T \mathbf{K}(-\nabla p + \rho_n \mathbf{g}) > 0. \\ \lambda_n(1 - s_r), & \text{otherwise.} \end{cases} \end{aligned} \tag{6.26}$$

Insertion of (6.25) and (6.26) into (6.23) yields an implicit system

$$\begin{aligned} &\sum_{i\in\mathcal{N}(j)} A_{j,i}\Big((\lambda_w^*)_{j,i} + (\lambda_n^*)_{j,i}\Big)\, \mathbf{n}_{j,i}^T \mathbf{K}(-\nabla p_{j,i}) = \\ &\qquad (Q_w)_j + (Q_n)_j - \sum_{i\in\mathcal{N}(j)} A_{j,i}\Big((\lambda_w^*)_{j,i}\rho_w + (\lambda_n^*)_{j,i}\rho_n\Big)\, \mathbf{n}_{j,i}^T \mathbf{K}_{j,i}\mathbf{g}. \end{aligned} \tag{6.27}$$

To solve (6.27), sam(oa)^2 implements a staggered scheme that alternates between computing the upstream mobilities λ_w^* and λ_n^* and solving for the pressure p until convergence is reached. The solver starts with an initial guess of p, and first computes (6.26). It then feeds the results into (6.27) to construct a linear system and solves it for p. With the updated p, the solver computes (6.26) once again and compares the results with the previous values. If λ_w^* and λ_n^* changed, meaning that convergence is not reached, the solver must perform further iterations and the upstream mobilities must be tested again. Until λ_w^* and λ_n^* no longer change, the solver returns and the solution of p is found.

Computation of the Time Step Size

The *CFL condition*, named after Courant, Friedrich and Lewy [110, 111], is a stability condition for the time step size Δt. It states that for each cell $j \in \{1, \ldots, n\}$

$$\left| \Delta t \sum_{i \in \mathcal{N}(j)} \frac{v_{j,i}}{\Delta x_{j,i}} \right| \leq 1, \tag{6.28}$$

where Δt is the global time step size, $v_{j,i}$ is the signal velocity at the interface between cell pair j, i, and $\Delta x_{j,i}$ is the characteristic length of the interface, which is defined as the ratio of the effective cell volume to the interface area, i.e., $\Delta x_{j,i} = \dfrac{\Phi_j V_j}{A_{j,i}}$ for the porous media flow model. Moving Δt to the left side and everything else to the right, we get

$$\Delta t \leq \frac{\Phi_j V_j}{\left| \displaystyle\sum_{i \in \mathcal{N}(j)} A_{j,i} v_{j,i} \right|}. \tag{6.29}$$

The signal velocity $v_{j,i}$ in the porous media flow model is indeed the maximum speed $\xi^-_{j,i}$ of all incoming waves at the interface of j, i given by (6.21).

Due to the fact that $\xi^-_{j,i}$ is difficult to compute, sam(oa)^2 implements a simpler condition that directly forces the saturation s_j in (6.22) to remain in the range of $[0, 1]$, i.e.,

$$0 \leq s_j + \frac{\Delta t}{\Phi_j V_j} \Big((Q_w)_j - \sum_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}_w(s_j, s_i) \Big) \leq 1, \tag{6.30}$$

which results in the condition of

$$\Delta t \leq \max \left(- \frac{\Phi_j V_j s_j}{(Q_w)_j - \displaystyle\sum_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}_w(s_j, s_i)}, \; \frac{\Phi_j V_j (1 - s_j)}{(Q_w)_j - \displaystyle\sum_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}_w(s_j, s_i)} \right). \tag{6.31}$$

At each IMPES iteration, Δt takes the maximal value that satisfies (6.31).

Adaptive Mesh Refinement

Adaptive mesh refinement requires three ingredients: an error indicator for refinement or coarsening, an interpolation operator for cell splitting (refinement) and a restriction operator for cell merging (coarsening). These three components must be defined for each quantity of interest, i.e., saturation s and pressure p in the porous media flow simulation.

For the error indicator for saturation, the $\mathcal{L}_1$ norm, given by

$$\mathcal{L}_1(s) = \int_\Omega |s_{\text{exact}} - s| \, \mathrm{d}\Omega, \tag{6.32}$$

is a suitable candidate, because it mitigates peaks at discontinuities that exist in the analytical solution s_{exact}. Assuming that s is an $\mathcal{L}_1$-optimal piecewise constant discretization of s_{exact}, the error in the $\mathcal{L}_1$ norm is bounded by

$$\mathcal{L}_1(s) \leq \frac{1}{2} \sum_{j=1}^{n} \big| \max_{x \in \Omega_j}(s_{\text{exact}}) - \min_{x \in \Omega_j}(s_{\text{exact}}) \big| \, V(\Omega_j), \tag{6.33}$$

which suggests that the degrees of freedom is invested well in cell j. This in sam(oa)^2 is modeled by

$$\Delta s_j V(\Omega_j) := |\max_{x\in\Omega_j}(s) - \min_{x\in\Omega_j}(s)|\, V(\Omega_j), \tag{6.34}$$

with the full refinement indicator given by

$$\Delta s_j V(\Omega_j) > \mathrm{Tol}_s\,(s_{\max} - s_{\min})\, V(\Omega_{\min}), \tag{6.35}$$

where $\mathrm{Tol}_s > 0$ is a dimensionless parameter chosen at runtime, $s_{\max} - s_{\min} = 1$ represents the maximum saturation difference, and $V(\Omega_{\min})$ denotes the minimal cell volume.

For the error indicator for pressure, sam(oa)^2 implements the following

$$\Delta p_j > \mathrm{Tol}_p\,(p_{\max} - p_{\min}) \frac{\Delta x_{\min}}{x_{\max} - x_{\min}}, \tag{6.36}$$

where Δx is the size of a cell, and $x_{\max} - x_{\min}$ represents the domain size. It is meant to be defined in a way such that Tol_p is independent of the problem, which is the reason why it includes the global value of pressure drop-off $p_{\max} - p_{\min}$ and the domain size.

Conservation of Mass, Pressure, Porosity and Permeability

For simulation of the SPE10 benchmark scenario, in which correct global quantities such as accumulated oil production are required, conservation of mass, pressure, porosity and permeability must be ensured for adaptive mesh refinement.

For mass, point-wise conservation $m_\alpha = \rho_\alpha \Phi s_\alpha$ is not possible due to the fact that refinement and coarsening modify the shape of cells and thus inevitably change the saturation in some regions. However, element-wise conservation can be achieved by using finite element theory. Let shape function $\hat{\phi}$, porosity $\hat{\Phi}$ and saturation $\hat{s}_\alpha$ be a set of variables living on the old grid, and a corresponding set ϕ, Φ and s_α living on the new grid. Let $\{\psi_j\}$ be a set of test basis functions. Element-wise mass conservation yields

$$\int_\Omega \rho_\alpha \Phi_j \Big(\sum_i (s_\alpha)_i \phi_i\Big) \psi_j \,\mathrm{d}\Omega = \int_\Omega \rho_\alpha \hat{\Phi}_j \Big(\sum_i (\hat{s}_\alpha)_i \hat{\phi}_i\Big) \psi_j \,\mathrm{d}\Omega, \tag{6.37}$$

from which mass conservative interpolation and restriction operators for s can be defined. We skip the implementation detail here. Simply put, the transfer of saturation s from the old grid to the new grid is achieved by computing the saturation of each cell on the new grid by averaging all saturations of cells on the old grid, weighted by the intersecting volumes of water in old and new cells, i.e.,

$$s_j := \frac{\sum_{i=1}^{\hat{n}} V(\hat{\Omega}_i \cap \Omega_j) \hat{s}_i}{V(\Omega_j)}. \tag{6.38}$$

For pressure, refinement and coarsening should keep the linear system in a solved state. However, after refinement (splitting of cells), there is no local flux-conservative solution for the new elements. Therefore, the pressure equation must be solved globally after each adaptive mesh refinement. For the pressure solver, sam(oa)^2 chooses an initial value p_j on the new grid by weighting the old pressure $\hat{p}$ with intersection volumes of old and new cells $V(\hat{\Omega}_i \cap \Omega_j)$ and averaging over the new cell area, i.e.,

$$p_j := \frac{\sum_{i=1}^{\hat{n}} V(\hat{\Omega}_i \cap \Omega_j) \hat{p}_i}{V(\Omega_j)}. \tag{6.39}$$

Strict mass conservation requires volume-weighted averaging porosity refinement. After a cell refinement or coarsening, the porosity of the coarse cell should be equal to the average porosity of both refined cells. For the case of coarsening, porosity in the new grid is computed straightforward by averaging. For refinement, the condition can be fulfilled by integrating the porosity data on the fly. Refinement and coarsening of the permeability are performed in the same manner as for porosity.

The Main Simulation Loop

Putting together all the necessary building blocks, the major steps of the porous media flow simulation are summarized in Algorithm 6.1. The application consists of three phases: *initialization* and *computation* and *finalization*.

In the *initialization* phase, the algorithm first creates a grid along with all the grid-associated data p, s, $\mathbf{K}$ and Φ. Then it refines the initial grid and interpolates p, s, $\mathbf{K}$ and Φ until the user-defined maximal refinement level is reached. As already discussed, after every mesh refinement, the pressure equation must be solved. The staggered scheme for alternating between computing upstream mobility and solving for pressure until convergence is wrapped in a loop, as shown in line 9 to 12.

The *computation* phase contains one main loop, which implements the IMPES scheme. At each iteration, the application first performs a traversal for setting the refinement flags for each cell. It then performs the actual grid refinement and coarsening, interpolates/restricts p, s, $\mathbf{K}$ and Φ accordingly, and follows it by a load balancing step, which is similar to the procedures discussed in Section 5.2.3. A while loop for a staggered pressure solver is executed and the solution for p is obtained. Then the time step size is computed according to (6.31). Saturation s is updated with an explicit time step given by (6.22). Lastly, visualization outputs are produced every few iterations.

6.3 Resource-elastic Transformation

In this section, we discuss how to transform the parallel porous media flow simulation shown in Algorithm 6.1 into a malleable application with the Elastic MPI library. The simulation is communication-intensive and implements an SPMD model. It contains one main simulation loop, in which resource adaptivity should take place. The initial grid refinement loop in the *initialization* phase has little computation intensity, hence will not be considered for resource adaptivity. Using Algorithm 4.2 as a guideline, the malleable parallel porous media flow simulation is summarized in Algorithms 6.2 and 6.3.

In terms of elastic programming models, this application does not bring in more variety as it has a computation workflow very similar to that of the tsunami simulation discussed in Chapter 5. However, as mentioned earlier, this application is interesting and worthwhile investigating due to its different parallel performance characteristics, i.e., being communication-bound and having a static workload, which cause the application to behave differently to resource adaptivity.

The Main Function

The key in Elastic MPI programming is to distinguish the different types of processes and handled them differently. In the main function, identifying the `JOINING` processes and

Algorithm 6.1: Main algorithm of the oil reservoir simulation from sam(oa)^2

```
1  Function Main():
2      MPI_Init()
3      Initialize numRanks, rank, etc.
       // 1. Initialization
4      Traversal: Initialize p, K and Φ
5      Traversal: Initialize s, set refinement flags, set up linear system of p
6      Traversals: Solve for p
7      while refinement flags are set do                    // grid initialization loop
8          Traversal: Adapt grid (interpolate p, s, K, Φ) and balance load
9          while pressure equation is not solved do
10             Traversal: Initialize s, set refinement flags, set up linear system of p
11             Traversals: Solver for p
12         end
13     end
14     t ← 0
       // 2. Computation: IMPES simulation main loop
15     while t < t_max do
16         Traversal: Set refinement and coarsening flags
17         Traversal: Adapt grid (interpolate p, s, K, Φ) and balance load
18         while pressure equation is not solved do
19             Traversal: Initialize s, set refinement flags, set up linear system of p
20             Traversals: Solver for p
21         end
22         Traversal: Compute time step Δt
23         Traversal: Update s
24         t ← t + Δt
25         if every K iterations then
26             Write visualization output s, p
27         end
28     end
       // 3. Finalization
29     MPI_Finalize()
30 End
```

Algorithm 6.2: Main algorithm of the Elastic MPI oil reservoir simulation

```
1  Function Main():
2      MPI_Init_adapt()                       // returns process status in procStatus
3      Initialize numRanks, rank, etc.
       // 1. Initialization: JOINING processes bypass
4      if procStatus = JOINING then
5          Resource_Adapt()                   // loop counter t is synced
6      else
7          Traversal: Initialize p, K and Φ
8          Traversal: Initialize s, set refinement flags, set up linear system of p
9          Traversals: Solve for p
10         while refinement flags are set do   // grid initialization loop
11             Traversal: Adapt grid (interpolate p, s, K, Φ) and balance load
12             while pressure equation is not solved do
13                 Traversal: Initialize s, set refinement flags, set up linear system of p
14                 Traversals: Solver for p
15             end
16         end
17         t ← 0
18     end
       // 2. Computation: IMPES simulation main loop
19     while t < t_max do
20         Traversal: Set refinement and coarsening flags
21         Traversal: Adapt grid (interpolate p, s, K, Φ) and balance load
22         while pressure equation is not solved do
23             Traversal: Initialize s, set refinement flags, set up linear system of p
24             Traversals: Solver for p
25         end
26         Traversal: Compute time step Δt
27         Traversal: Update s
28         t ← t + Δt
29         if every K iterations then
30             Write visualization output s, p
31         end
32         if Time to probe Resource Manager then
33             MPI_Probe_adapt()              // returns adapt decision in adaptFlag
34             if adaptFlag says to adapt then
35                 Resource_Adapt()
36             end
37         end
38     end
       // 3. Finalization
39     MPI_Finalize()
40 End
```

Algorithm 6.3: Resource adaptation of the Elastic MPI oil reservoir simulation

```
32 Function Resource_Adapt():
33     MPI_Comm_adapt_begin()
34     if there are LEAVING processes then
           // No need to load balance as it occurs at next iteration beginning
35         LEAVING processes send their sections to STAYING processes
36     end
37     if there are JOINING processes then
38         JOINING processes create and initialize grid and data objects
39         Sync data (loop counters t, etc.)
40     end
       // Data redistribution is omitted as it occurs at next iteration beginning
41     MPI_Comm_adapt_commit()               // MPI_COMM_WORLD is updated
42     Update numRanks, rank, procStatus, etc.
43 End
```

routing them correctly is a crucial step. Due to the intentional latency hiding design, preexisting processes are only notified of the arrival of `JOINING` processes by calling the `MPI_Probe_adapt` function. And the two groups of processes can only start collaborating when they reach the beginning of the adaptation window `MPI_Comm_adapt_begin`. Due to this reason, the `JOINING` processes must be directed to the designated `Resource_Adapt` function as soon as possible bypassing the *initialization* phase, because they are meant to receive proper data preparation inside the adaptation window.

In Algorithm 6.2, the *initialization* phase is wrapped in a conditional such that it is only executed by the original processes that are created during program start. Processes joining the program at runtime are directly routed to the `Resource_Adapt` function shown in Algorithm 6.3. Upon returning from the adapt function, the `JOINING` processes enter the main simulation loop (line 19) and merge with the preexisting processes.

The main simulation loop remains mostly the same as in the original program (Algorithm 6.1), except for an additional conditional placed at the end of the loop, i.e., line 32 to 37. This is the preexisting processes' entry point to the `Resource_Adapt` function. The conditional is for controlling the frequency of probing the resource manager. Ideally, the probing should happen frequently enough such that it does not miss any resource change decisions, yet not too often such that it creates unnecessary overhead. More importantly, the adaptation frequency should coincide with the application's inherent load balancing frequency in order to minimize overhead, because data redistribution is required after every resource adaptation. This condition is met in the current application, since a load balancing step already takes place at every iteration.

Resource Adaptation

A resource adaptation only takes place if `MPI_Probe_adapt` returns a positive signal. Inside the adaptation window, there are three possible types of processes: `STAYING`, `JOINING` and `LEAVING`. In the current Elastic MPI implementation, only pure resource expansion and pure reduction are supported.

In case of reduction, the `LEAVING` processes send their grid sections to the staying processes. Load imbalance is not a concern at this point, because a load balancing step will take place in the beginning of the next iteration. In case of expansion, the `JOINING` processes initialize a grid and the necessary data structures, i.e., s, p, $\mathbf{K}$ and Φ. The loop counter t is synchronized between the preexisting and `JOINING` processes. Data redistribution step is omitted also because a load balancing step will be performed in the next iteration.

6.4 Performance Evaluation

In this section, we present the performance and resource efficiency analysis on the SPE10 benchmark scenario described in Section 6.1. Simulations were run with the oil field porosity and permeability data provided by the benchmark. We conducted tests to determine the impact of resource adaptivity on the application itself. We also compared performance between Elastic MPI and static MPI runs to find out whether resource efficiency could be improved.

6.4.1 Execution Environment: Virtual Machine Emulated Cluster

Due to limited access to the SuperMUC Petascale System at the time this thesis was written, all tests with this application were conducted on a virtual machine (VM) emulated cluster. As previously mentioned, the Elastic MPI infrastructure is designed for general HPC systems. It can be installed on conventional Linux clusters with GNU compiler support, transforming the cluster into a resource-elastic environment.

The VM cluster consists of 8 compute nodes, 1 controller node and 1 login node, which are all hosted on a single machine. Table 6.1 lists some important specifications of the VM cluster as well as the host machine. The controller node and the login node do not participate in computation. The controller node is where the elastic resource manager is located, and the login node merely serves the purpose of letting users log on to the cluster and launch applications. Each MPI process is pinned to a CPU. As each compute node has 2 CPUs, the VM cluster provides from minimally 2 processes (1 node) to maximally 16 processes (8 nodes).

Table 6.1: Specifications of the VM cluster and its host machine

	Host Machine	VM Cluster
Processor	Intel Kaby Lake Core i7-7700T Quad	Generic
Nominal frequency [GHz]	3.6	2.5 (capped at 70% of host)
Cores per node	4	2
Memory per node [GByte]	32	2
Number of nodes	1	8+2
Interconnect	–	TCP/IP

6.4.2 The Benchmark: Reduced SPE10 Scenario

Due to the limited computing power of the VM cluster, full-sized simulations of the SPE10 benchmark (such as the one displayed by Figure 6.2) are not feasible. To reduce the computational intensity such that results could be obtained within a reasonable time frame, we reduced the simulation domain from its full size of 85 vertical layers to the bottom 16 layers, and shortened the simulated time from 730 days (2 years) to 400 days.

Figure 6.4 demonstrates the saturation profile visualization (left), the top view of the underlying adaptive 3-D grid (right) as well as the visualization of resource utilization (middle) of a reduced SPE10 benchmark simulation with 16 layers at two different time steps. The color blocks in the grid correspond to the color of the executing nodes, e.g., the blue portion of the grid is computed by the node in the same color. Nodes in gray color mean they are not utilized. Figure 6.4(a) shows 1 node being utilized at simulated day 100, and Figure 6.4(b) shows 8 nodes being utilized at day 400.

The major concerns of running tests on the VM cluster are that the simulation domain must be truncated, and that the performance of the application is impaired. Fortunately for our case, neither concern is imperative, because

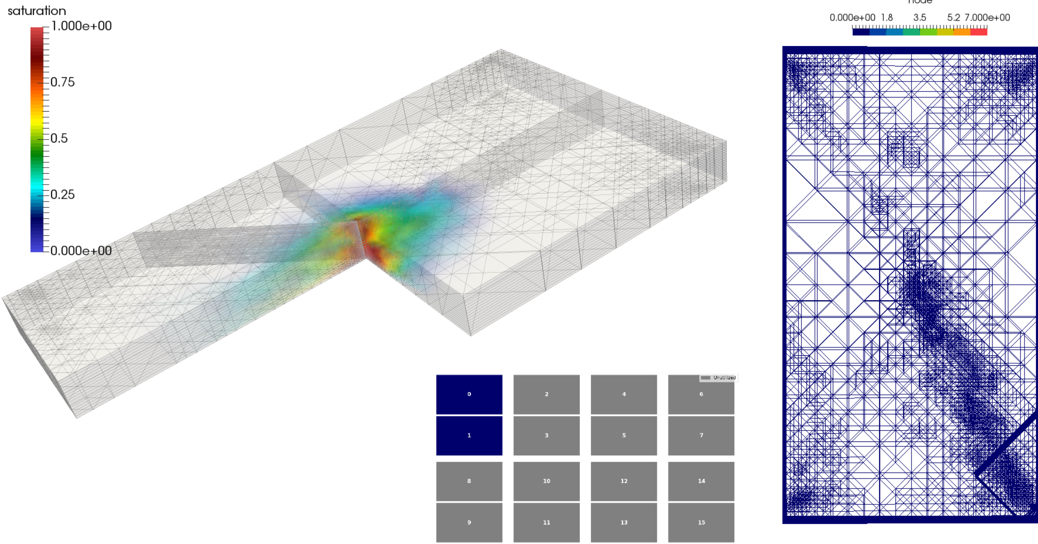
1. we do not aim to obtain high fidelity simulation results, but rather checking for the correctness of the simulation with added resource adaptivity, to which 16 layers would do justice;
2. we are not interested in comparing performance with other benchmarks or across platforms, but rather in the performance comparison between Elastic MPI and static (normal) MPI runs in identical execution environments, therefore, it does not matter in which environment the tests are conducted as long as the environment is kept unchanged across tests.

6.4.3 Resource Adaptivity Overhead

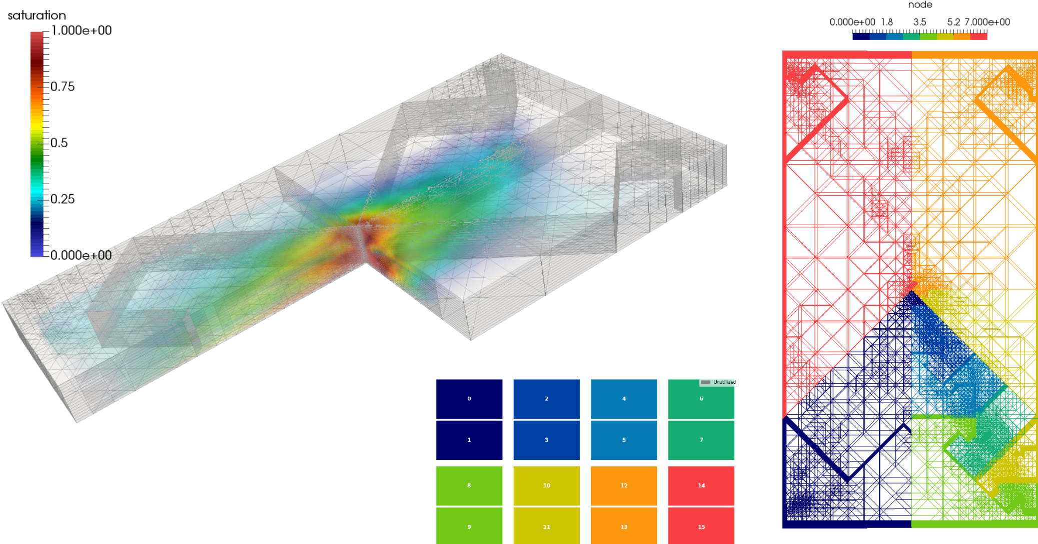
Resource adaptivity poses concerns on introducing significant overhead from not only additional function calls but also frequent data repartitioning and redistribution. We want to analyze the impact of resource adaptation on the oil reservoir simulation. We conduct several tests with a single instance of the simulation in an 8-node elastic environment. In each test, the simulation always starts with minimal resources – 1 node (2 processes). The maximal resources it can expand to is 8 nodes (16 processes).

In order to fully examine different cases of resource expansion and reduction, we use a random elastic scheduler that gives new resource assignment randomly between 1 and 8 nodes every 10 minutes. There is no adaptation if the generated random number is the same as the current number of nodes. The application is set to probe the resource manager every 500 simulation steps.

Figure 6.5 displays plots of the MPI processes profile and the number of grid cells of one test run. The blue curve with the left y-axis depicts the change of computational resources (number of MPI processes) over time. The red curve with the right y-axis illustrates how the computational workload (number of grid cells) changes as the simulation evolves in time. The x-axis in Figure 6.5(a) shows the simulated time in days. The x-axis in Figure 6.5(b) shows the execution time (elapsed time) in hours. In this run, 18 adaptations, including 10 expansions and 8 reductions, occurred during a total runtime of 3.2 hours.

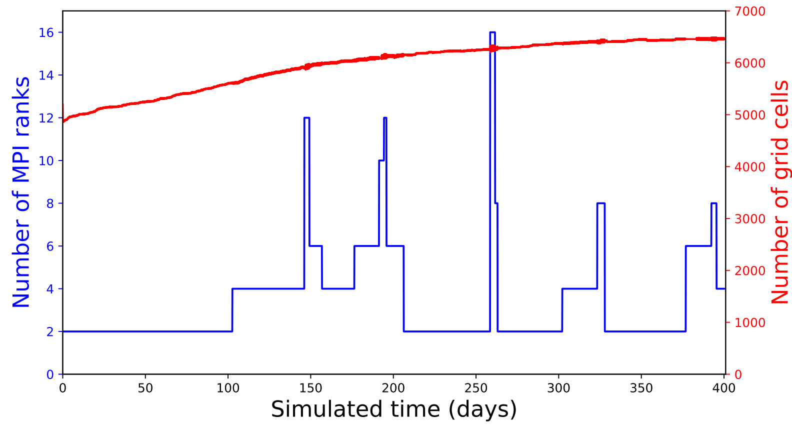


(a) Saturation profile, underlying grid and resource utilization at simulated day 100

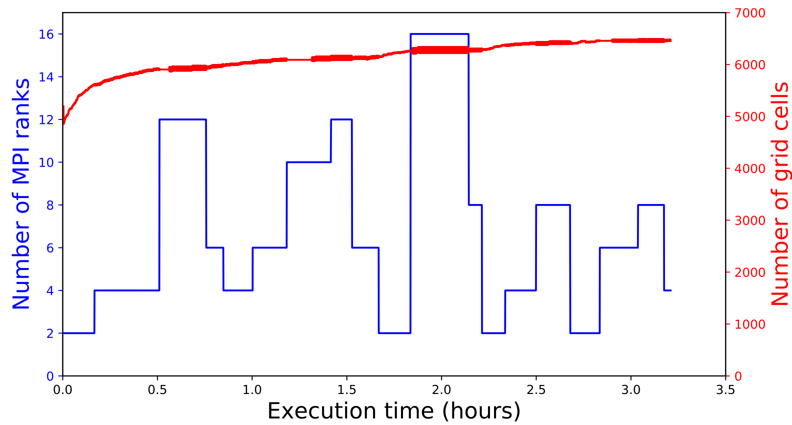


(b) Saturation profile, underlying grid and resource utilization at simulated day 400

Figure 6.4: Reduced SPE10 simulation with 16 vertical layers with Elastic MPI on VM cluster. The saturation profile is visualized on the left. Top view of the underlying adaptive 3-D grid is displayed on the right. Resource utilization is visualized in the middle. The color blocks in the grid correspond to the color of the executing nodes, e.g., the blue portion of the grid is computed by the node in the same color. Nodes in gray color mean they are not utilized. In this test run, 1 node is utilized at simulated day 100, and 8 nodes are utilized at day 400.



(a) # of MPI processes, # grid cells vs. simulated time (days)



(b) # of MPI processes, # grid cells vs. execution time (hours)

Figure 6.5: The computational resources (number of MPI processes) and workload (number of grid cells) profile of an Elastic MPI test run with a random scheduler. The upper figure plots the profiles against simulated time in days. The lower figure plots the profiles against execution time in hours.

Table 6.2 summarizes the execution time of Elastic MPI function calls. The second column shows the average execution time of each function, the third column shows the accumulative execution time, and the last column shows the percentage of the function's accumulated execution time takes up in the total adaption time. The last row in this table shows the total time the application spent on data movement within the adaptation window. In normal cases data migration should be the most dominant item, but this application is an exception since its actual data redistribution and load rebalancing operations take place outside of the adaptation window.

The simulation is decomposed into major computational tasks, and Table 6.3 summarizes the execution time spent on each of them. The item we are most interested in is *resource adaptation*, which takes up 4.5 seconds out of a total execution time of 11545 seconds (about 3.2 hours). This leads to the conclusion that the overhead introduced by resource adaptivity is trivial.

Table 6.2: Execution time of Elastic MPI functions

Elastic MPI function	Avg. time (sec)	Acc. time (sec)	% of total adap.
`MPI_Init_adapt`	0.0500	0.0500	1.11%
`MPI_Probe_adapt`	0.0009	0.2074	4.60%
`MPI_Comm_adapt_begin`	0.0827	1.4889	33.04%
`MPI_Comm_adapt_commit`	0.1100	1.9805	43.95%
Data migration	0.0433	0.7789	17.29%

Table 6.3: Execution time of computational tasks

Computational task	Exec. time (sec)	% of total exec. time
Total simulation	11545.43	100.00%
Time step computation	6391.40	55.36%
Grid refinement	1602.46	13.88%
Grid conformity check	1318.00	11.42%
Load balancing	1539.46	13.33%
Resource adaptation	4.51	0.04%
Others	689.86	5.98%

In addition to the adaptation overhead, two important pieces of information can be extracted from this test run from Figure 6.5:

- **The change in computational workload is little.** For grid based simulations, the number of grid cells is the most important indicator for the computational workload. Unlike the tsunami simulation in Chapter 5, which has a drastic growth in grid cell number of 6 times from simulation start to end, this simulation has a much stabler grid cell count.

  Figure 6.5(b) shows that starting from the 15-minute mark until the end of the simulation, the grid cell number only grows by about 8% from 6000 to 6500 cells. The almost constant workload is a clear indication that the application does not have a resource change requirement, and that the best fitting resource profile would be constant.

- **Performance of this application is better with fewer resources.** Knowing that the computational workload has little influence on performance, we can analyze the correlation between number of MPI processes and performance. Comparing Figure 6.5(a) with Figure 6.5(b), we can see that the the application progresses much faster with fewer processes. For instance, about 100 simulated days are computed in the first 10 minutes with 2 processes, whereas with 16 processes, more than 18 minutes are needed to compute 5 simulated days.

  To demonstrate this more clearly, we provide another plot in Figure 6.6, which shows the relationship between the average execution time of simulation steps and number of MPI processes. The application spends much less time (performs better) on computing a simulation step with less processes. It performs best with 2 processes. This number of course is specific to the current configuration of the simulation and it might change if the simulation had a different workload, e.g., if it had 85 layers instead

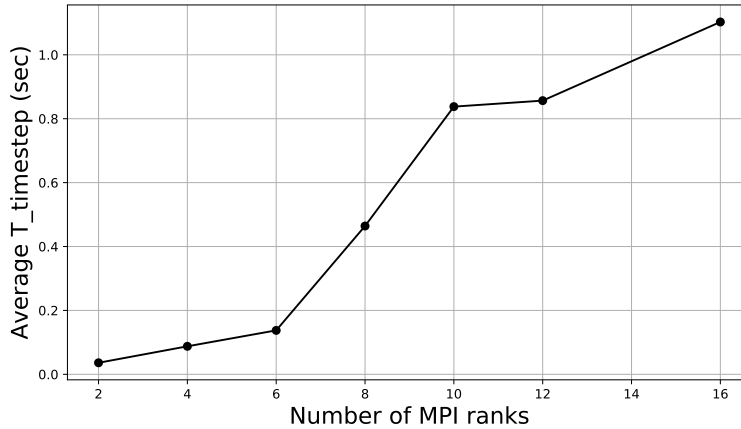


Figure 6.6: Average execution time of computing one step vs. Number of MPI processes.

of 16. A possible explanation for this can be that the application is communication-bound and sensitive to communication overhead, therefore, more resources means more communication overhead thus leads to poorer performance.

6.4.4 Runtime and Resource Efficiency

In this experiment, we want to compare performance and resource efficiency of tests with Elastic MPI and static MPI on the 8-node VM cluster. Because the performance monitoring functionality and the decision logic in the resource manager are not fully implemented, our Elastic MPI tests are conducted with pre-defined runtime schedulers.

We make three test runs:

1. **Static MPI with 16 processes**: In this test, the simulation runs with the standard MPICH library with 16 processes. The choice of utilizing all available resources on the cluster is natural, assuming no prior information on the application's scalability.
2. **Elastic MPI with stepping scheduler**: Based on previous experiments, we know that the computational workload (number of grid cells) increases throughout the simulation, even though only mildly. Therefore, we design the second test run with a *stepping* elastic scheduler, which increases the resource assignment by 1 node in every adaptation until it reaches the maximum node count limit. The blue curve in Figure 6.7 shows the resource profile in this test run, and the red curve shows the workload profile.
3. **Elastic MPI with 2 processes unchanged**: This test is devised to mimic the ideal scenario that through continuous performance monitoring, the resource manager learns that the application performs best with 2 processes and decides not to change the resource assignment throughout the simulation.

Figure 6.8 shows the plotting of the resource profiles (number of MPI processes) of all three test runs against execution time in hours. Since MPI processes are pinned to CPUs, the resource utilization (in CPU hours) of each run can be computed by integrating its resource profile curve over its execution time. The CPU hours can be visualized by the

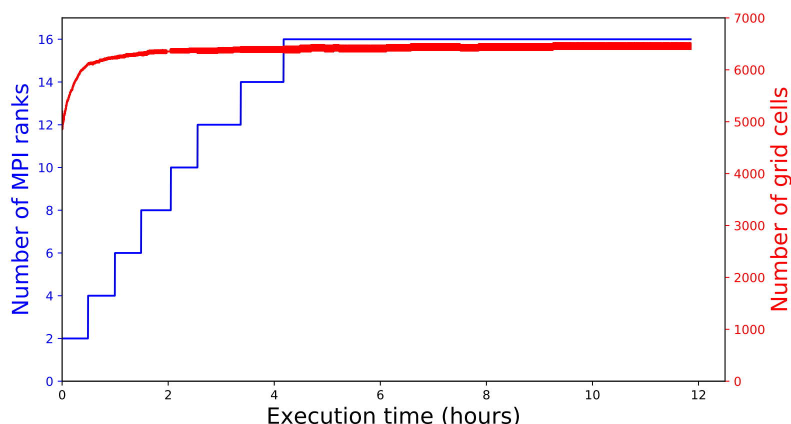


Figure 6.7: The computational resources (number of MPI processes) and workload (number of grid cells) profile of an Elastic MPI test run with a stepping scheduler.

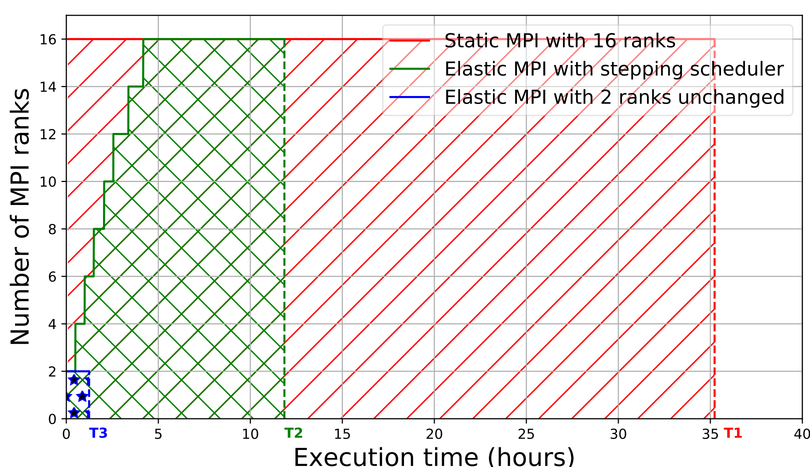


Figure 6.8: Resource profile vs. execution time and CPU hours of three test runs.

shaded area under each resource profile curve. The exact execution time and CPU hours of each run are further listed in Table 6.4.

The first run with static MPI at 16 processes has the worst performance. It took over 35 hours to complete resulting in 562 CPU hours. Comparing to the first run, the second run with Elastic MPI and a stepping scheduler shows clear performance improvement. Its execution time is reduced to 34%, and the CPU hours are also reduced to 28%. The third run with Elastic MPI and an unaltered resource assignment at 2 processes is the clear winner in both runtime and resource efficiency. Comparing to the static MPI run, its runtime is reduced to 3% and CPU hours are reduced to less than 1%.

To further examine the performance quality of the three test runs, we decompose the simulation into major tasks and analyze their computational effort distribution, i.e., we do not compare the actual execution time, but the percentage of the total execution time spent on each task.

From Table 6.5 we can see that test runs 1 and 2 have very similar distribution. They both spend close to 45% time on time step computation, close to 25% time on grid opera-

Table 6.4: Execution time and CPU hours of three test runs

Test run	Exec. time (hours)	CPU hours
1. Static MPI with 16 processes	35.23	562.31
2. Elastic MPI with stepping scheduler	11.85	159.39
3. Elastic MPI with 2 processes unchanged	1.24	2.47

Table 6.5: Execution time distribution of three test runs

Computational task	1. Static MPI with 16 processes	2. Elastic MPI with stepping scheduler	3. Elastic MPI with 2 processes unchanged
Total simulation	100.00%	100.00%	100.00%
Time step computation	43.04%	44.04%	77.36%
Grid refinement	10.07%	11.38%	16.44%
Grid conformity check	13.35%	13.43%	2.82%
Load balancing	22.80%	21.28%	0.68%
Resource adaptation	0.00%	0.01%	0.01%
Others	10.74%	9.86%	2.70%

tions (*grid refinement* and *grid conformity check*), and more than 20% on communication (*load balancing*). Their computation-communication ratio is roughly 2 : 1.

In comparison, the third run has significant performance improvement with 77% on time step computation and less than 1% on communication, which translates into a computation-communication ratio of over 110 : 1. Its time on grid operations is also slightly reduced to about 19%, which is still similar to that of the other two runs. Notice that even though resource adaptation did not occur in this run, the execution time on *resource adaptation* is non-zero due to function calls to `MPI_Init_adapt` and `MPI_Probe_adapt`.

6.5 Summary

We selected the 3-D oil reservoir simulation from the sam(oa)^2 framework as our second case study for malleable software development with Elastic MPI. This is yet another classical communication-intensive grid-based HPC application implementing a SPMD model. Despite having a very similar computational workflow as the tsunami simulation, this application demonstrated very different behaviors with resource adaptivity.

Due to limited access to SuperMUC at the time this thesis was written, we had to conduct tests for this application on an 8-node VM cluster with 2 CPUs per node. Determined by the limited computing power of the host machine, we had to size down the simulation benchmark by truncating the domain from 85 to 16 layers in the vertical dimension. We were able to do so because the main goal of our tests was not to obtain high accuracy simulation results, but to check the correctness with added resource adaptivity and to compare performance between static and elastic runs in an identical execution environment.

Resource adaption overhead was shown to have minimal impact on performance in all test cases. This was mainly due to the fact that the application had a built-in load balancing

scheme. By overlapping these built-in operations with those required by resource adaptation, we were able to avoid performing data redistribution during the resource adaption windows. Another reason has to do with the appropriate setting for resource adaptation frequency. On an hours-long runtime scale, adapting once every few minutes is shown to be a good practice.

Unlike the tsunami simulation whose computational workload is highly dynamic, this oil reservoir simulation had a comparably stable workload, i.e., its number of grid cells did not vary much, which was a clear indication that it did not require runtime resource changes. The test results confirmed that a stable resource assignment, when fitted to the application's scalability, brought optimal performance and resource efficiency.

Due to its communication-bound characteristics and limited workload, the application was prone to communication overhead and therefore, performed better with less MPI processes. For such applications, the naive way of utilizing as many available resources as possible is not the best idea. The test case of running with the maximal 16 MPI processes showed an extreme case of poor performance and resource inefficiency. The optimal resource assignment (2 processes), in comparison to the worst case (16 processes), reduced not only the overall execution time to three percent, but also the CPU hours to less than one percent.

The most fundamental question is that: can Elastic MPI help to improve performance and resource efficiency for applications with constant workload? The answer is definitely positive. Even though such an application might not require runtime resource adjustment, finding its optimal resource assignment still remains a task to be fulfilled. The Elastic MPI framework is equipped with continuous performance monitoring (even though it is currently not fully implemented), and runtime resource decisions are made according to elastic applications' real-time scalability. Only when an application is Elastic MPI enabled, can it be monitored throughout runtime and provided with optimal resource assignments.

Statistical Inverse Problem Solver with Elastic Parallel Surrogate Construction

Inverse problems are some of the most important mathematical problems due to their ubiquity in many fields. They typically involve recovery of hidden (unobservable) quantities or reconstruction of models based on indirect observations, for example, calculating the density of the earth from measurements of its gravity field, location of oil and mineral deposits from seismic data and well logs, creation of astrophysical images from telescope data, image reconstruction in computer vision, source reconstruction in acoustics, modeling in life sciences, and many more.

In this chapter, we present a solver for high dimensional inverse problems with the help of a surrogate model. The most computationally intensive part of the problem solving process is the surrogate model construction, for which a resource-elastic parallel implementation is introduced. This application is chosen for Elastic MPI investigation for two reasons:

1. It represents an important class of scientific problems that are frequently encountered in many fields and very challenging to solve.
2. It provides diversity and completeness to our application base, because surrogate construction in this application is an embarrassingly parallel problem. It implements a master-worker execution model, and has multiple resource-adaptive phases with a different workload in each phase.

In the following sections, we first introduce a statistical approach for solving inverse problems. Then we discuss an adaptive surrogate model construction method using sparse grid interpolation and its implementation with the Elastic MPI library. For a case study, we solve the problem of locating obstacles in a fluid channel using the discussed methods and tools, and conduct performance analysis on the impact of resource adaptivity.

7.1 Statistical Inverse Problems

In this section, we discuss a statistical approach, more specifically, the Bayesian approach for solving inverse problems. In the Bayesian framework, an inverse problem is reformulated as an inference problem in which the goal is to update the probability for a hypothesis given the observation as evidence. A short introduction to the Markov Chain Monte Carlo method, a popular solver often employed in Bayesian inference problems, is also presented.

7.1.1 Inverse Problems

In a normal simulation problem, given a model that simulates a physical phenomenon, one computes simulation output from a set of input parameters. Such process can be expressed by

$$\vec{y} = f(\vec{x}), \tag{7.1}$$

where $f : \Omega \to \Theta$ represents the simulation model – a mapping relating input parameters to simulation output, $\vec{x} \in \Omega$ a set of input parameters, and $\vec{y} \in \Theta$ the simulation output. For a given set of parameters, there is a corresponding set of simulation output, which is the unknown of interest (marked in red color). In the inverse-problem terminology, this process is referred to as the *forward problem*, and model f is referred to as the *forward model*, because the process works from the input to output.

In an inverse problem, on the contrary, the process works in the opposite direction: one calculates the causal factors (the input parameters to the forward model) based on some observed data (the output of the model), hence the name *inverse*. This process can be expressed by

$$\vec{y}_{\text{observed}} = f(\vec{x}) + \eta, \tag{7.2}$$

where $\vec{y}_{\text{observed}}$ is given, and $\vec{x}$ is the unknown to solve for (marked in red color). Noise is always present due to uncertainties in the simulation model as well as error from observation or measurement. In (7.2), noise is modeled as an additive term represented by η.

Deterministic Solution

Due to the fact that in most cases, there is no direct mapping from the output to input, i.e., $\vec{x} = f^{-1}(\vec{y})$, solving an inverse problem relies on performing many forward simulations with different plausible inputs and comparing all the simulation outputs with the observation [112].

A classical strategy for solving inverse problems is to use the *regularization theory*, in which one defines a *loss function* to quantify the differences between the observation and the simulation output, and a *regularizer* to impose physical constraints and bounds, or encode prior information of the solution, and defines the solution as the one that minimizes the sum of these two terms [113], i.e.,

$$\vec{x}_* = \arg\min_{\vec{x} \in \Omega} \Big[\mathcal{L}(\vec{y}_{\text{observed}}, f(\vec{x})) + \mathcal{R}(\vec{x}) \Big]. \tag{7.3}$$

This approach produces a single optimal solution that fits the observation best under given constraints, therefore, is also referred to as an *optimization-based approach*.

Methods fall into the regularization category are well studied. They address the issues from the ill-posedness of the inverse problems. However, they fall short of accounting for uncertainties from data observations/measurements and the simulation model. On the other hand, statistical approaches are capable of handling uncertainties in different stages of the problem solving process, thanks to their stochastic nature.

7.1.2 Bayesian Inference Framework

In the Bayesian interpretation, probability is considered as a degree of belief conditional to prior assumptions and experience of the observer, and could be updated in the presence of

new evidence. This reformulates the inverse problem with conditional probabilities: what is the probability for the input parameters to take on certain values given the observed data as evidence? This approach leads to a solution of a probability density distribution of x over Ω.

Modeling the output data, input parameters and noise as random variables (represented by capital letters), their relationship can be expressed by

$$Y = f(X) + H, \tag{7.4}$$

where $X \in \Omega$ and $Y \in \Theta$. Modeling noise as an additive term ensures that the probability distribution of H remains unaltered regardless of the value of X.

Let $\pi_A(a)$ be the probability for random variable A takes on value a. According to Bayes' theorem [114], the solution to problem (7.2) can be formulated as

$$\pi_{XY}(x \,|\, \vec{y}_{\text{observed}}) = \frac{\pi_{\text{prior}}(x)\ \pi_{XY}(\vec{y}_{\text{observed}} \,|\, x)}{\pi_Y(\vec{y}_{\text{observed}})}, \tag{7.5}$$

assuming that prior knowledge of the input parameters could be encoded into a probability distribution $\pi_{\text{prior}}(x) := \pi_X(x)$, and that the probability distribution of the observation is positive, i.e., $\pi_Y(\vec{y}_{\text{observed}}) = \int_\Omega \pi_{XY}(x, y)\,\mathrm{d}x > 0$.

Omitting the Marginal Likelihood

The *marginal likelihood* $\pi_Y(\vec{y}_{\text{observed}})$ is also referred to as the *model evidence*, because it is an indicator of the relative confidence level of the forward model. It is a constant for all possible values of X, as X does not appear in it. Therefore, when not assessing the uncertainty of the model, solution to (7.2) can be simplified as

$$\pi_{\text{posterior}}(x) := \pi_{\text{prior}}(x)\ \pi_{XY}(\vec{y}_{\text{observed}} \,|\, x), \tag{7.6}$$

where $\pi_{\text{posterior}}(x) \propto \pi_{XY}(x \,|\, \vec{y}_{\text{observed}})$.

Construction of the Likelihood

The term $\pi_{XY}(\vec{y}_{\text{observed}} \,|\, x)$ is called the *likelihood.* When fixing the unknown to a certain value, e.g., $X = x$, it can be deduced that the probability of Y conditioned on $X = x$ equals the probability distribution of H, i.e.,

$$\begin{aligned} \pi_{XY}(y \,|\, x) &= \pi_{XY}(f(x) + \eta \,|\, x) \\ &= \pi_X(f(x) + \eta \,|\, f(x)) \\ &= \pi_{\text{noise}}(\eta) \\ &= \pi_{\text{noise}}\Big(\vec{y}_{\text{observed}} - f(x)\Big), \end{aligned} \tag{7.7}$$

where π_{noise} represents the probability distribution of noise. Substituting (7.7) into (7.6), the solution can be expressed as

$$\pi_{\text{posterior}}(x) := \pi_{\text{noise}}\Big(\vec{y}_{\text{observed}} - f(x)\Big)\ \pi_{\text{prior}}(x). \tag{7.8}$$

More details on deriving the Bayesian inference solution can be found in [113].

7.1.3 Markov Chain Monte Carlo Methods

The computation of $\pi_{\text{posterior}}(x)$ over Ω relies on sampling from Ω and computing the probability at each sample according to (7.8). The computation for every sample involves performing a forward simulation, i.e., $f(x)$.

The challenge is that most practical inverse problems are high-dimensional, i.e., $\Omega \subset \mathbb{R}^M$ with $M \gg 1$, which means a lot of samples are required, thus an equal amount of forward simulations. With isotropic sampling schemes, the computation effort grows exponentially with the number of dimensions. In order to break the *curse of dimensionality* [115], random sampling methods are often employed.

Markov Chain Monte Carlo (MCMC) methods are typically used in Bayesian inferences due to two desired properties: one, they rely on repeated random sampling, which decouples dimensionality of the parameter space from the sampling process; and two, they ensure convergence to a desired distribution, i.e., $\pi_{\text{posterior}}(x)$, if properly constructed.

In depth discussions on the theories of MCMC methods can be found in numerous literature as a standalone topic or part of a statistical context, such as [116, 117, 118, 113, 119], to name a few. A full-length discussion on the MCMC theory are not provided in this section, however, the key concept of *Markov Chain* as well as the algorithms that are implemented in the application are presented. For the rest of this section, all concepts, theorems and algorithms discussed can be found in most MCMC literature such as the ones mentioned above.

Markov Chain

In the context of sampling, a first order Markov chain is defined to be a series of random variables $\{X_i \in \Omega : i = 1, \dots, N\}$ distributed according to $p(x)$ such that the following property holds for all $n \in \{1, \dots, N-1\}$

$$p(X_{n+1} = x_{n+1} \,|\, X_1 = x_1, \dots, X_n = x_n) = p(X_{n+1} = x_{n+1} \,|\, X_n = x_n). \tag{7.9}$$

The series of random variables are the *states* that the chain goes through. In a first order Markov chain, the next state is dependent only on the current state.

The probability distribution that transfers the chain from one state to the next is called the *transition kernel*, denoted by $T(x_n, x_{n+1})$. It is indeed the conditional probability for the subsequent variable, i.e.,

$$T(x_n, x_{n+1}) \equiv p(x_{n+1} \,|\, x_n). \tag{7.10}$$

A distribution $p(x)$ is said to be *invariant* with respect to a Markov chain if and only if

$$\int_\Omega T(x, y)\, p(x)\, \mathrm{d}x = p(y). \tag{7.11}$$

With an invariant distribution, the transition kernel of the Markov chain does not depend on the state index, i.e.,

$$T(x_n, x_{n+1}) = T(x_{n+k}, x_{n+k+1}), \tag{7.12}$$

in which case the chain is said to be *stationary* or *converged.* Invariant distributions are also referred to as *stationary* or *equilibrium distributions*.

In Bayesian inference problems, we want to employ Markov chains to sample from the posterior distribution, which can be achieved by choosing a Markov chain whose invariant distribution equals to the posterior distribution. Therefore, we are interested in under what conditions a Markov chain converges and how to ensure that it converges to the posterior distribution. Listed below are some concepts related to convergence:

- **Irreducibility**: a Markov chain is *irreducible* if

  $$T^k(x_i, x_j) > 0 \ \ \forall\, i, j \in \{1, \dots, N\}\,, \quad k < \infty,$$

  where k represents the number of transition steps. Simply put, a Markov chain is irreducible if it is possible to get to any state from any state in a finite number of steps. This can be satisfied if

  $$\forall y : p(y) > 0 \rightarrow T(x, y) = p(y \,|\, x) > 0 \, \forall x.$$

- **Aperiodicity**: the period of a state x is defined as

  $$d(x) := \text{g.c.d.}\{k \geq 1 : T^k(x, x) > 0\},$$

  where g.c.d. denotes the greatest common divisor, and k the number of transition steps. A chain is *aperiodic* if $d(x) = 1 \ \forall x$. In sampling, this is usually satisfied because the states are random variables subjected to stochastic processes, hence not periodic.

- **Reversibility**: a Markov chain is *reversible* if its transition kernel satisfies the *detailed balance* condition given by

  $$p(x_n)\, T(x_n, x_{n+1}) = p(x_{n+1})\, T(x_{n+1}, x_n)$$

  Reversibility implies that the chain is stationary. It is indeed a sufficient but not necessary condition for the chain being $p(x)$-invariant.

- **Convergence Theorem**: for any irreducible and aperiodic Markov chain,
  - there exists at least one invariant distribution (existence).
  - there exists exactly one invariant distribution (uniqueness).
  - let $\pi(x)$ be the invariant distribution, and $\pi_0(x)$ any initial distribution,

    $$\pi_n(x) \xrightarrow{n \to \infty} \pi(x) \ \ \text{(ergodicity)}.$$

  The theorem simply states that if a Markov chain is irreducible and aperiodic, it is ensured to converge to a unique invariant distribution regardless of its initial state.

Metropolis-Hastings Algorithm

In Bayesian inference, we want to draw samples from the posterior distribution. This can be achieved by constructing a Markov chain that is ensured to converge to the posterior distribution. The first successful attempt was the *Metropolis Algorithm*, proposed by N. Metropolis in 1953. A generalized version was later developed, called the *Metropolis-Hastings Algorithm* (MH), which is summarized in Algorithm 7.1. It can be shown that the Markov chain in MH is irreducible and aperiodic and that the transition kernel satisfies the detailed balance condition, guaranteeing convergence towards $\pi(x)$.

Algorithm 7.1: Metropolis-Hastings Algorithm

Data: $\pi(x)$ the target posterior distribution
Data: $q(a \mid b)$ any proposal distribution
Data: x_n the state of the Markov chain at transition step n
Result: Samples $\{x_1, \dots x_N\}$

1 Initialize x_1
2 **for** $n = 1$ **to** N **do**
3 Draw a candidate sample z from $q(z \mid x_n)$
4 Calculate acceptance ratio:

$$a(x_n, z) = \min\left\{1, \frac{\pi(z)}{\pi(x_n)} \frac{q(x_n \mid z)}{q(z \mid x_n)}\right\}$$

5 Update the next state by setting

$$x_{n+1} = \begin{cases} z & \text{with probability } a(x_n, z) \\ x_n & \text{with probability } 1 - a(x_n, z) \end{cases}$$

6 Set $x_n = x_{n+1}$.
7 **end**

In the inverse problem presented in this chapter, a serial MCMC solver is implemented according to Algorithm 7.1 with a normal proposal distribution and a single-dimension updating strategy. The proposal distribution is given by

$$q(z \mid x_n) = \mathcal{N}(x_n, \sigma),$$

where σ, usually referred to as the *random walk size*, is empirically chosen to be 5% of the domain size of the problem. Single-dimension updating strategy means that only one dimension (component) of $X \in \mathbb{R}^M$ is updated at each transition step, e.g.,

$$\text{given } x_n = \begin{bmatrix} x_1^{(n)} \\ \vdots \\ x_i^{(n)} \\ \vdots \\ x_M^{(n)} \end{bmatrix}, \quad x_{n+1} = \begin{bmatrix} x_1^{(n)} \\ \vdots \\ x_i^{(n+1)} \\ \vdots \\ x_M^{(n)} \end{bmatrix}, \quad \text{where } i \in \{1, \dots, M\}.$$

Parallel Tempering Algorithm

The MH is the most basic form of MCMC methods. It is a serial process that theoretically guarantees convergence to a target distribution. However, MH is not very practical because it falls short of discovering multimodal target distributions within a reasonable amount of transition steps. For most practical inverse problems, the posterior distribution is unknown, and so unimodality is simply not guaranteed. The plain-vanilla MH is likely to be trapped in one of the modes when the target is multimodal with isolated modes, failing to discover the true posterior distribution.

Algorithm 7.2: Parallel Tempering Algorithm

Data: $\pi(x)$ the target posterior distribution
Data: K number of independent MH chains
Data: $\pi_i(x)$ the invariant distribution of chain i, given by

$$\pi_i(x) = \pi(x)^{\frac{1}{T_i}}, \quad \text{where } \; i \in 1, \ldots, K, \quad 1 = T_1 < T_2 < \cdots < T_K$$

Data: $x_i^{(n)}$ the state of chain i at transition step n, $i \in 1, \ldots, K$
Result: Samples from chain 1: $\{x_1^{(1)}, \ldots, x_1^{(N)}\}$

1 Initialize $\{x_1^{(1)}, \ldots, x_K^{(1)}\}$, start K independent MH chains in parallel
2 **for** $n = 1$ **to** N **do**
3   **Update operation:**
      Each chain i update its next state $x_i^{(n+1)}$ using the MH kernel, e.g., execute line 3 – 5 of Algorithm 7.1.
4   **Exchange operation:**
      At a certain frequency (not necessarily every step), select two chains g and h
      Chain g computes exchange acceptance rate

$$e_g = \min \left\{ 1, \; \pi(x_g^{(n+1)})^{\frac{1}{T_h} - \frac{1}{T_g}} \right\}$$

      Chain h computes exchange acceptance rate

$$e_h = \min \left\{ 1, \; \pi(x_h^{(n+1)})^{\frac{1}{T_g} - \frac{1}{T_h}} \right\}$$

      Swap samples $x_g^{(n+1)}$ and $x_h^{(n+1)}$ with probability $e_g \cdot e_h$
5   Each chain i set $x_i^{(n)} = x_i^{(n+1)}$
6 **end**

The *Parallel Tempering Algorithm* (PT) is an improved MCMC variation to overcome the above mentioned deficiency by generating candidate samples from all over the distribution. The general idea of PT is to run K independent MH chains in parallel, with each chain i equipped with an invariant distribution $\pi_i(x)$, which is a smoothened version of the original target $\pi(x)$ with a controlling parameter T_i for smoothness, i.e.,

$$\pi_i(x) = \pi(x)^{\frac{1}{T_i}}, \quad i \in \{1, \dots, K\}.$$

T_i is often referred to as the *temperature* of $\pi_i(x)$. The higher the T_i, the smoother (closer to uniform) the $\pi_i(x)$ is compared to the original target $\pi(x)$, thus, the easier it is for chain i to move between modes. The temperatures are often defined in a ladder form such as $1 = T_1 < T_2 < \cdots < T_K$, with which only the first chain is sampling from the original target, i.e., $\pi_1(x) = \pi(x)$.

Exchange of samples between two chosen chains (usually adjacent ones) are periodically proposed and executed with a certain acceptance rate. These exchanges push samples from the hotter chains (with higher temperatures) to colder ones (with lower temperatures), which eventually helps the coldest chain (the first chain) to escape from isolated modes. The final result of a PT consists of only samples from the first chain. The PT implemented in this application is summarized in Algorithm 7.2.

7.2 Surrogate Model Construction with Sparse Grids

Aside from high dimensionality, another challenge encountered in most practical inverse problems is high complexity of the forward model, which means carrying out a forward simulation is computationally expensive and time consuming. With an MCMC solver, we obtain the solution by asymptotically sampling from the posterior distribution. Each sample requires computation of the posterior probability given by (7.8), which in turn requires a forward simulation. The total computational cost for solving an inverse problem is determined by the cost for running a forward simulation multiplied by the number of forward simulations.

In order to reduce computational costs or just to make the computation feasible, *surrogate models* are often employed. Surrogate models (or surrogates) are approximations to the original complex, high-fidelity models with much less computation intensity but also some trade-offs on accuracy. In this section, we discuss a surrogate model construction algorithm based on function interpolation with adaptive sparse grids. An elastic parallel implementation of the algorithm is presented in the end.

7.2.1 Function Interpolation

There exist many methods for surrogate model construction, such as projection-based reduced-order methods, hierarchical methods, data-fitting methods, among others. Function interpolation falls under the data-fitting class, which produces a model approximation using interpolation or regression from the simulation output of the original model. It does not require modification or manipulation to the original model kernel, instead, simply treats it as a black-box. Such *non-intrusive* property makes the method very attractive, especially in cases where access to the model kernel is not available.

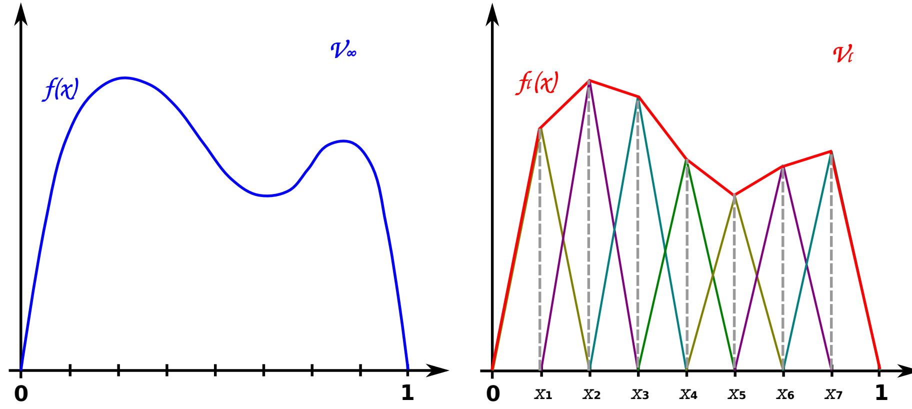


Figure 7.1: 1-D function interpolation on unit interval

The general idea of constructing surrogates with function interpolation is to find a proper representation of the original model on a coarser (lower resolution) function space, which is linearly spanned by a finite set of basis functions. This is illustrated by a 1-D example shown in Figure 7.1.

Let $f : [0,1] \to \mathbb{R}$ be a smooth function defined on function space $\mathcal{V}_\infty$, and $\mathcal{V}_l$ a function space spanned by a finite set of basis functions $\{\phi_i(x) \,|\, i = 1, \ldots, 2^l - 1\}$, i.e.,

$$\mathcal{V}_l = \text{span}\{\phi_i\} = \left\{ \sum_{i=1}^{2^l-1} c_i \phi_i(x) \right\}. \tag{7.13}$$

In the example shown, $l = 3$ and $\mathcal{V}_3$ is spanned by $\{\phi_1, \ldots, \phi_7\}$, which is a set of piecewise linear hat functions defined on equidistant grid points $\{x_i = \frac{i}{2^3} \,|\, i = 1, \ldots, 7\}$ over the unit interval. Note that basis functions can be chosen arbitrarily and are not limited to hat or other linear functions. The approximation of $f(x)$ on $\mathcal{V}_l$ is given by

$$f(x) \approx f_l(x) = \sum_{i=1}^{2^l-1} c_i \phi_i(x). \tag{7.14}$$

Since the basis functions are known (chosen), finding $f_l(x)$ is equivalent to finding the values of the coefficients $\{c_i\}$, which can be obtained by evaluating $f(x)$ at the grid points, i.e.,

$$c_i = f(x_i),\ \forall i = 1, \ldots, 2^l - 1. \tag{7.15}$$

7.2.2 Sparse Grids

Sparse grids are mathematical structures originally developed for spatial discretization for solving partial differential equations. There have been successful applications of sparse grids in other problem areas such as integration, interpolation, approximation, data mining, among others. Full length discussions on the sparse grid theory is out of scope of this section, but they can be found in many dedicated literature such as [120, 121, 122, 123, 124, 125, 126, 127], to name a few. Construction of sparse grids are based on a hierarchical basis and a sparse tensor product construction. The general idea is demonstrated in Figure 7.2 with a 2-D example.

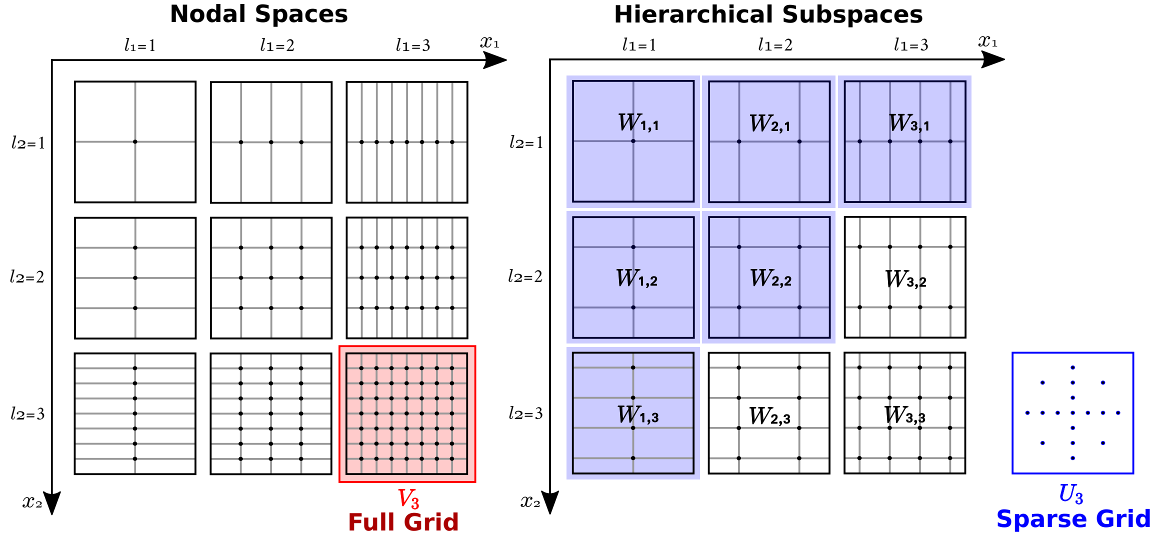


Figure 7.2: A 2-D isotropic grid $\mathcal{V}_3$ with discretization level = 3, its hierarchical decomposition and its corresponding sparse grid $\mathcal{U}_3$

Let $\Omega = [0,1]^d$ be a unit hypercube and multi-index $\vec{l} = [l_1, \ldots, l_d] \in \mathbb{N}^d$ a d-dimensional discretization scheme. Applying $\vec{l}$ on Ω results in an anisotropic grid discretized with mesh

$$h_{\vec{l}} := [h_{l_1}, \ldots, h_{l_d}] = [2^{-l_1}, \ldots, 2^{-l_d}].$$

Let $\mathcal{V}_{\vec{l}}$ denote a function space supported by this grid, that is, it is spanned by a finite set of basis functions defined on each grid point,

$$\mathcal{V}_{\vec{l}} := \text{span}\{\phi_{\vec{j}} \,|\, \vec{j} = [j_1, \ldots, j_d],\ j_k = 0, \ldots, 2^{l_k},\ k = 1, \ldots, d\}.$$

$\{\phi_{\vec{j}}\}$ can be arbitrarily chosen. For simplicity, let them be a set of piecewise d-linear hat functions.

Let $\mathcal{V}_l$ denote an isotropic space with uniform discretization in every dimension, i.e., $l_1 = \cdots = l_d = l$. We call the supporting grid for $\mathcal{V}_l$ the *full grid* of level l. The example shown in Figure 7.2 is a 2-D case with $d = 2$, $l = 3$. $\mathcal{V}_3$ (highlighted in red) can be hierarchically decomposed into a set of 9 subspaces $\{\mathcal{W}_{\vec{l}}\}$ as shown to the right, i.e.,

$$\mathcal{V}_3 = \bigoplus_{l_1, l_2 = 1}^{3} \mathcal{W}_{l_1, l_2}. \tag{7.16}$$

Details on *hierarchical decomposition* can be found in most sparse grid literature such as the ones mentioned earlier.

According to the *cost-contribution analysis* of hierarchical subspaces [120, 121, 122, 123], the cost-contribution ratio of a subspace $\mathcal{W}_{\vec{l}}$ is dependent on a constant given by

$$|\vec{l}|_1 := \sum_{k=1}^{d} l_k. \tag{7.17}$$

A smaller value of $|\vec{l}|_1$ indicates a smaller cost-contribution ratio. With the goal of minimizing cost while maximizing contribution, it is possible to construct an approximation

Table 7.1: Comparison of number of grid points of $l = 5$ full grids and sparse grids

d	$\mathcal{V}_5$	$\mathcal{U}_5$
1	31	31
2	961	129
3	29,791	351
4	923,521	769
5	28,629,151	1,471
6	887,503,681	2,561
7	27,512,614,111	4,159
8	852,891,037,441	6,401
9	26,439,622,160,671	9,439
10	819,628,286,980,801	13,441

to $\mathcal{V}_l$ by selecting only those subspaces that have smaller cost-contribution ratio than the others. It is shown that the optimal selection is given by

$$\{\vec{l} : |\vec{l}|_1 \leq l + d - 1\}. \tag{7.18}$$

In the demonstrated example in Figure 7.2, the optimal selection includes all subspaces with $|\vec{l}|_1 \leq 4$ (highlighted in blue). A direct combination of the optimal selection leads to a space $\mathcal{U}_l$ that can well approximate $\mathcal{V}_l$ with less basis supports. The supporting grid of $\mathcal{U}_l$ is called a sparse grid.

The *size* of a grid refers to its total number of grid points, which represents the computation complexity of the structure. Sparse grids are good tools for mitigating the curse of dimensionality, because unlike full grids, whose sizes have exponential dependency on dimensionality d, i.e., $\mathcal{O}(2^{l \cdot d})$, sparse grids have sizes on the order of $\mathcal{O}(2^l \cdot l^{d-1})$, which are far smaller than those of their corresponding full grids. Table 7.1 enumerates the size differences between level 5 full grids $\mathcal{V}_5$ and their sparse grid counterparts $\mathcal{U}_5$ with different dimensionality. The computational superiority of sparse grids becomes more evident as dimensionality increases.

7.2.3 Function Interpolation on Adaptive Sparse Grids

There have been successful applications of sparse grids on function interpolation. It is shown that when the interpolating function meets certain smoothness condition [125, 128, 120, 121, 122, 123], the error of its sparse grid interpolant is bounded. However, very often there are cases in which the interpolating function does not meet such requirements, e.g., when it comprises both very steep and flat regions. In these cases, adaptive sparse grids are required.

A standard sparse grid is one that is constructed by combining the optimal selection of the hierarchical subspaces, i.e., the ones satisfying condition (7.18) as discussed in Section 7.2.2. Such a grid approximates an isotopic grid with the same discretization resolution in every dimension. However, sparse grids can also be constructed adaptively to provide different resolutions in different dimensions or localized regions. To be dimension-adaptive [123], a sparse grid can include higher resolution subspaces in certain dimensions

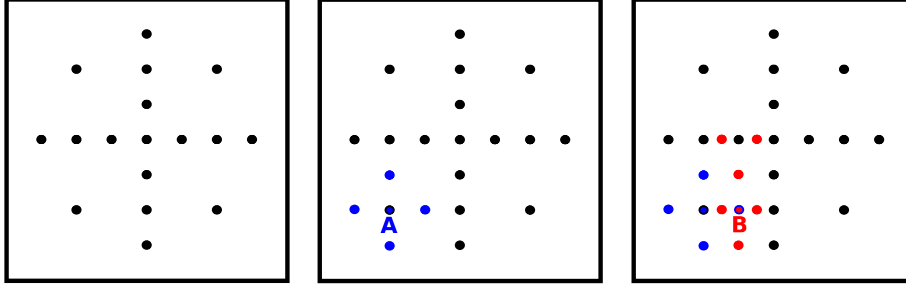


Figure 7.3: 2-D adaptive sparse grid

in addition to the set of optimal selection. To be spatially adaptive, it can refine on selected grid points (local regions), which is called *adaptive refinement* [122].

Figure 7.3 demonstrates two adaptive refinement steps on a 2-D level 3 sparse grid, which is first refined on grid point A as shown in the middle figure, and then further refined on grid point B as shown in the right figure. Due to their ability to capture more details or information where necessary (depending on the refinement criteria), adaptive sparse grids can tackle functions that do not meet the general smoothness condition by adapting to their special characteristics. The surrogate construction algorithm implemented for the inverse problem discussed in this chapter employs function interpolation on sparse grids with adaptive refinement.

Similar to the 1-D case discussed in Section 7.2.1, interpolating a model $f(x)$ on a d-dimensional sparse grid is equivalent to finding the coefficients $\{\alpha_{\vec{j}}\}$ in

$$f(x) \approx f_{\text{SGI}}(x) = \sum_{\vec{j} \in \mathbf{B}^P} \alpha_{\vec{j}} \phi_{\vec{j}}(x), \tag{7.19}$$

where $\mathbf{B}^P$ is the set of indices of all P grid points in the sparse grid, and $\{\phi_{\vec{j}}\}$ a set of chosen basis functions each located at a grid point. In our implementation, we choose d-linear hat functions as basis.

Due to the fact that basis functions in sparse grids are hierarchical basis (more details on *hierarchical decomposition* [120, 121, 122, 123]), coefficients $\{\alpha_{\vec{j}}\}$ are not direct evaluations of $f(x)$ at the grid points, which is different from the 1-D case. But they can be computed from these values, and the process is called *hierarchization*. Let $x_{\vec{j}}$ denote the coordinate of the grid point at which $\phi_{\vec{j}}(x)$ is located,

$$\{\alpha_{\vec{j}}\} = \texttt{hierarhize}(\{c_{\vec{j}}\}), \tag{7.20}$$

where $c_{\vec{j}} = f(x_{\vec{j}})\ \forall \vec{j} \in \mathbf{B}^P$. $\{\alpha_{\vec{j}}\}$ are ofter referred to as the *hierarchical surpluses.*

In short, the computation of hierarchical surpluses $\{\alpha_{\vec{j}}\}$ consists of P data-independent subtasks of evaluating $f(x)$ at each grid point as well as an aggregation step for hierarchization. This means that the construction of surrogate is an embarrassingly parallel computation problem.

In the implemented algorithm, the surrogate is initially constructed on a standard sparse grid of a certain level specified by the user. A model error of the surrogate is then computed over a set of randomly selected test points $\{t_i \in \Omega \,|\, i = 1, \ldots, Q\}$. It is the average Euclidean distance between the output from the original model and that from the surrogate,

i.e.,

$$e = \frac{1}{Q} \sum_{i=1}^{Q} \| f(t_i) - f_{\text{SGI}}(t_i) \| . \tag{7.21}$$

While the surrogate model error e is greater than a set threshold, i.e., while the surrogate is not accurate enough, the sparse grid keeps refining in regions of rapid changes and high posterior probabilities, because rapid changes indicate poor resolution for capturing the model details, and high posterior probabilities indicate areas of higher interests.

A refinement indicator is computed for every grid point $x_{\vec{j}}$ as

$$r_{\vec{j}} = \left\| \alpha_{\vec{j}} \right\| \cdot \pi_{\text{posterior}}(x_{\vec{j}}). \tag{7.22}$$

The $l2$-norm of the hierarchical surpluses is used because it is a great indicator for the gradient magnitude. In a refinement, a user-specified number of grid points with the greatest r values are refined, and new grid points are added to the grid.

The computational cost of a refinement is the number of added grid points multiplied by the cost of running a forward simulation. Let $C(f)$ be the cost of running a forward simulation with the original model $f(x)$, P_0 the number of grid points in the initial sparse grid, P_i the number of added grid points in the ith refinement, and R the total number of refinement performed, then the total cost of constructing a surrogate is given by

$$C_{\text{SGI}} = C(f) \times (P_0 + \sum_{i=1}^{R} P_i).$$

7.3 Case Study: Inference of Obstacle Locations in Laminar Flow

In this section, we discuss an inverse problem in which we try to locate multiple obstacles in a fluid channel based on measurements of the fluid velocity. The solution is formulated with Bayesian inference. An MCMC solver based on the parallel tempering algorithm is implemented. The solver is coupled with a surrogate model constructed with function interpolation on adaptive sparse grids. Construction of the surrogate is the most computationally intensive part of the entire problem solving workflow. A malleable parallel algorithm for surrogate construction based on Elastic MPI is presented.

7.3.1 The Problem

Consider a 2-D channel with a rectangular domain of 10 meters by 2 meters filled with incompressible viscous fluid. The fluid flows into the channel from the left boundary and flows out from the right boundary, as shown in Figure 7.4. Inside the channel there are four obstacles with known sizes and shapes, but unknown locations. Ten sensors are placed across the channel to obtain measurements of the fluid velocity at four time instances, which produces a total number of 40 (10×4) measurements.

We have a model $f : \Omega \to \Theta$ that simulates the fluid dynamics. It takes the locations of four obstacles as input parameters, and outputs velocities of the fluid at the exact locations and times at which the sensors make measurements. Since the simulation domain is 2-D, the location of each obstacle consists of an x- and a y-coordinate, the input parameter

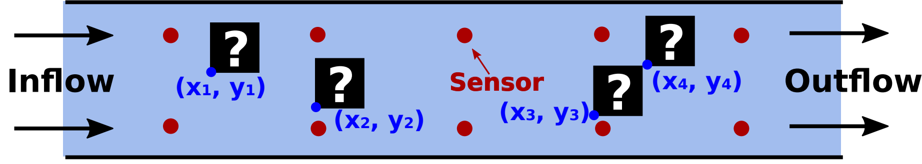


Figure 7.4: A 2-D fluid channel with four obstacles at unknown locations

therefore consists of 8 components, i.e., $\vec{x} = [x_1, y_1, \ldots, x_4, y_4] \in \Omega \subset \mathbb{R}^8$, which means the inverse problem is 8-dimensional.

Assuming that there is Gaussian noise in the data measurements, i.e.,

$$\pi_{\text{noise}}(x) = \mathcal{N}(0, \sigma^2 I),$$

and that we have no prior information of the obstacle locations, i.e., the prior distribution is uniform,

$$\pi_{\text{prior}}(x) = \mathcal{U}_\Omega = c \in \mathbb{R},$$

we can further simplified the solution given by (7.8) by eliminating the constant prior term and obtain

$$\pi_{\text{posterior}}(x) := \exp\left(-\frac{1}{2\sigma^2}\left[\vec{y}_{\text{observed}} - f(\vec{x})\right]^T\left[\vec{y}_{\text{observed}} - f(\vec{x})\right]\right). \tag{7.23}$$

7.3.2 The Forward Simulation Model

The fluid dynamics of the incompressible flow can be described by the 2-D *Navier-Stokes equations* [104, 105, 129] given by

$$\begin{aligned}
\frac{\partial u}{\partial t} + \frac{\partial (u^2)}{\partial x} + \frac{\partial (uv)}{\partial y} &= -\frac{\partial p}{\partial x} + \frac{1}{\text{Re}}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + g_x, \\
\frac{\partial v}{\partial t} + \frac{\partial (uv)}{\partial x} + \frac{\partial (v^2)}{\partial y} &= -\frac{\partial p}{\partial y} + \frac{1}{\text{Re}}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) + g_y, \\
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0.
\end{aligned} \tag{7.24}$$

The first two equations, called the *momentum equations*, describe the conservation of momentum, and the third one, called the *continuity equation*, describes the conservation of mass. In (7.24), x and y denote the horizontal and vertical dimensions of the domain, t the time evolved, u and v the velocities of the fluid in the x- and y-direction respectively, and p pressure of the fluid. $\text{Re} \in \mathbb{R}$ is a dimensionless quantity called the *Reynolds number*, which characterizes the stickiness (freedom of movement) of the fluid. A smaller Re value indicates a more viscous fluid. g_x and g_y denote external forces in the x- and y-direction respectively, e.g, gravity or other body forces acting throughout the bulk of the system. The unknowns to be solved for are u, v and p.

Spatial Discretization

To solve the PDE system in (7.24), we employ the *staggered grid* [129] for spatial discretization. It is essentially a Cartesian grid, but with the unknowns placed at different locations in a grid cell, i.e., pressure p is placed at the center of a cell, the horizontal velocity u is

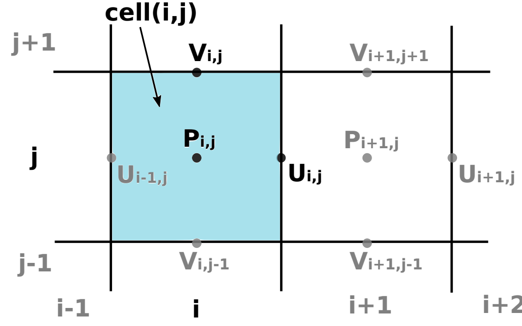


Figure 7.5: Staggered grid

placed at the right edge of the cell, and the vertical velocity v is placed at the top edge of the cell, as shown in Figure 7.5. With such scheme, each spatial derivatives can be approximated by a *finite difference* [130, 131] formula.

Time Discretization

For the momentum equations, we employ the *Explicit Euler* [132] scheme, which gives

$$\begin{aligned} u_{i,j}^{(n+1)} &= F_{i,j}^{(n)} - \frac{h_t}{h_x}\left(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}\right), \\ v_{i,j}^{(n+1)} &= G_{i,j}^{(n)} - \frac{h_t}{h_y}\left(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}\right), \end{aligned} \tag{7.25}$$

where $F_{i,j}^{(n)}$ and $G_{i,j}^{(n)}$ are given by

$$\begin{aligned} F_{i,j} &:= u_{i,j} + h_t\left(\frac{1}{\mathrm{Re}}\left(\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j}\right) - \left[\frac{\partial (u^2)}{\partial x}\right]_{i,j} - \left[\frac{\partial (uv)}{\partial y}\right]_{i,j} + g_x\right), \\ G_{i,j} &:= v_{i,j} + h_t\left(\frac{1}{\mathrm{Re}}\left(\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j}\right) - \left[\frac{\partial (uv)}{\partial x}\right]_{i,j} - \left[\frac{\partial (v^2)}{\partial y}\right]_{i,j} + g_x\right). \end{aligned} \tag{7.26}$$

From the continuity equation, we have

$$\left[\frac{\partial u}{\partial x}\right]_{i,j}^{(n+1)} + \left[\frac{\partial v}{\partial y}\right]_{i,j}^{(n+1)} = 0, \tag{7.27}$$

where, based on the spatial discretization finite difference scheme,

$$\begin{aligned} \left[\frac{\partial u}{\partial x}\right]_{i,j}^{(n+1)} &:= \frac{u_{i,j}^{(n+1)} - u_{i-1,j}^{(n+1)}}{h_x}, \\ \left[\frac{\partial v}{\partial y}\right]_{i,j}^{(n+1)} &:= \frac{v_{i,j}^{(n+1)} - v_{i,j-1}^{(n+1)}}{h_y}. \end{aligned} \tag{7.28}$$

Substituting (7.25) into (7.28), and in turn into (7.27) results in an equation from which pressure can be solved, i.e.,

$$\begin{aligned} &\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{h_x^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{h_y^2} \\ &\qquad = \frac{1}{h_t}\left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{h_x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{h_y}\right). \end{aligned} \tag{7.29}$$

Equation (7.29) is in the form of the discretized *Poisson's equation* [133, 134], and can be transformed into a linear system of

$$A\vec{p} = \vec{b}, \tag{7.30}$$

where matrix A reflects the discretized Laplace operator Δ, and b denotes the right hand side of (7.29).

To ensure stability of the numerical scheme, the following condition is imposed on the time step size h_t,

$$h_t := \tau \min \left\{ \frac{\mathrm{Re}}{2} \left(\frac{1}{h_x^2} + \frac{1}{h_y^2} \right)^{-1}, \frac{h_x}{|u_{\max}|}, \frac{h_y}{|v_{\max}|} \right\}, \tag{7.31}$$

where $|u_{\max}|$ and $|v_{\max}|$ are the maximal absolute values of the respective velocity, and the coefficient $\tau \in [0, 1]$ is a safety factor, usually set to 0.5. Condition (7.31) ensures that the fluid does not travel faster than one grid cell at a time step. The spatial and time discretization scheme employed in our implementation mainly follows [129].

Input and Output

To run a simulation of the fluid dynamics in the given 2-D channel is to solve the PDE system (7.24) with the right initial and boundary conditions. Obstacles inside the channels are treated as inner boundaries. Since their sizes and shapes are known, once their locations are set, a geometry with the proper domain and inner boundaries can be constructed, and the PDE system can then be solved.

The output of the simulation consists of all velocities and pressure values in each grid cell at each time step, i.e.,

$$\left\{ \begin{bmatrix} \vec{u}^{(t_0)} \\ \vec{v}^{(t_0)} \\ \vec{p}^{(t_0)} \end{bmatrix}, \begin{bmatrix} \vec{u}^{(t_1)} \\ \vec{v}^{(t_1)} \\ \vec{p}^{(t_1)} \end{bmatrix}, \dots, \begin{bmatrix} \vec{u}^{(t_{\mathrm{end}})} \\ \vec{v}^{(t_{\mathrm{end}})} \\ \vec{p}^{(t_{\mathrm{end}})} \end{bmatrix} \right\}.$$

For the inverse problem described in Section 7.3.1, not all simulation output data are necessary. Indeed, only the u and v values at the 10 locations and 4 time instances at which the sensors make the measures are kept.

In short, the forward simulation model $f : \Omega \to \Theta$ takes in a vector $\vec{x} \in \mathbb{R}^8$ that specifies the locations of the four obstacles, and produces a vector $\vec{y} \in \mathbb{R}^{40}$ of the fluid velocities sampled at 10 locations across the domain at 4 different time instances.

7.3.3 The Main Algorithm and Malleable Surrogate Construction

The main steps for solving the inverse problem is relatively straightforward:

1. build a surrogate model,
2. run the MCMC solver with the surrogate.

The output is a series of samples that represents the posterior distribution of the obstacle coordinates. For an adaptive surrogate construction, step 1. can be wrapped in a while loop such that the building process continues until the surrogate meets the accuracy requirement. The main function in Algorithm 7.3 consists of exactly these steps.

The most computationally intensive part of the program lies in the Build_Surrogate function, which on its own is a resource-adaptive computation phase in a master-worker style. The application has, therefore, indefinite computation phases. An elastic programming model for such type of applications is summarized in Section 4.5.

Surrogate Model Construction with Resource Adaptation

The Build_Surrogate function in Algorithm 7.3 either builds a surrogate model from scratch or refines an existing one by examining whether the input surrogate model argument f_{SGI} is empty or not. In either case, the process is similar:

1. a sparse grid is built or refined, resulting in a number of grid points being added;
2. then, the function enters a master-worker block to perform a forward simulation for each newly added grid point;
3. lastly, the function computes the hierarchical surpluses by hierarchizing the direct function evaluations $\{c_i\}$.

When JOINING processes arrive, they are routed to the Resource_Adapt block directly, bypassing the grid initialization/refinement block, because they will be prepared with the necessary grid-related data in the adaptation window. Upon returning from the Resource_Adapt function, they reach the master-worker block and start working as workers.

The Resource_Adapt function in Algorithm 7.3 opens an adaptation window. It handles the JOINING processes by preparing them with grid-related data and the surrogate model error err, which is the computation phase identifier. It omits the steps for handling LEAVING processes as well as those for data redistribution, such as lines 26-28 and 33 in Algorithm 4.2. This is because in this application, we use MPI-IO for saving and transferring data instead of MPI communication. When a grid is generated or refined, it is saved to a grid file by the Master. When a worker finishes a subtask, it saves the data to a data file, which eliminates the operations of migrating data from LEAVING processes.

The main reasons to opt for MPI-IO are memory consumption and parallel performance. The number of grid points can grow rapidly as more refinements are performed. Since every grid point $\vec{x}_i$ is associated with a long index i, a function evaluation output $c_i \in \mathbb{R}^{40}$ and a hierarchical surplus $\alpha_i \in \mathbb{R}^{40}$, with the long index being 64-bit and each double precision floating point being 64-bit, the memory required for storing a grid point is at least 648 Bytes. For $10,000$ grid points, the memory required to store the grid-related data is about 6.5 GB. If the grid grows larger in size, the compute node can be saturated or run out of memory. Besides concerns on memory consumption, when new processes join in the computation, the Master process must broadcast multi-gigabytes of data to all the JOINING processes, which is performance-wise impractical. Due to these reasons, the application is designed to use the file system as the data transfer medium. That is, all grid related data are written to files at the time they are produced, and can be retrieved by any process from these files at any time afterwards. With MPI-IO concurrent read and write[1], the data transfer process is significantly accelerated.

[1] Concurrent write is possible with MPI-IO if each process write to a non-overlapping subset of a file.

Algorithm 7.3: Main algorithm for locating obstacles in a fluid channel

```
1  Function Main():
2      MPI_Init_adapt()                          // returns process status in procStatus
3      Initialize numRanks, rank, etc.
       // 1. Surrogate construction
4      err ← Tol + 1
5      while err > Tol do
6          f_SGI ← Build_Surrogate(f, f_SGI)         // Initialize or refine surrogate
7          Compute, reduce, broadcast model error err
8      end
       // 2. MCMC solver
9      {x⃗_1, ..., x⃗_S} ← MCMCPT(S, f_SGI)
       // 3. Finalization
10     MPI_Finalize()
11 End

12 Function Build_Surrogate(f, f_SGI):
       // 1.1. Initialization: JOINING processes bypass
13     if procStatus = JOINING then
14         Resource_Adapt()                          // loop counter err is synced
15     else
16         if f_SGI is empty then
17             Initialize a standard sparse grid, added new grid points {x⃗_i};
18         else
19             Refine the underlying sparse grid, added new grid points {x⃗_i};
20         end
21     end
       // 1.2. Computation: evaluate f at each added grid points: c_i = f(x⃗_i) ∀i
22     if is Root then
23         Master()
24     else
25         Worker()
26     end
       // 1.3. Compute hierarchical surpluses collectively
27     {α_i} ← hierarchize({c_i})
28 End

29 Function Resource_Adapt():
30     MPI_Comm_adapt_begin()
31     if there are JOINING ranks then
32         JOINING ranks read grid data from file            // MPI-IO operations
33         Synchronize err among the new resource group
34     end
35     MPI_Comm_adapt_commit()                       // MPI_COMM_WORLD is updated
36     Update numRanks, rank, procStatus, etc.
37 End
```

Algorithm 7.4: Master and Worker functions for surrogate construction

```
1  Function Master():
       // The P grid points are divided into a number of jobs
2      Initialize jobsArray          // Track job status: Todo, Progress, or Done
3      Initialize workersArray               // Track worker status: Active or Idle
4      Seed workers
5      while not all jobs are Done do
6          if there are Active workers then
7              Receive a job done notification from any worker
8          end
9          if there are Todo jobs and there are Idle workers then
10             Send a Todo job to an Idle worker
11         end
           // Only make sense to adapt if there are more Todo jobs
12         if Time to probe Resource Manager then
13             MPI_Probe_adapt()             // returns adapt decision in adaptFlag
14             if adaptFlag says to adapt and there are Todo jobs then
15                 Check all workers status, collect job done notification from Active
                    workers
16                 Send adapt signal to all workers
17                 Resource_Adapt()
18                 Resize and update workersArray                // workers changed
19                 Seed workers
20             end
21         end
22     end
23     Send terminate signal to all workers
24 End

25 Function Worker():
26     while true do
27         Receive a message from Master
28         Extract Master's instruction from status object
29         if instruction is to terminate then
30             break
31         end
32         if instruction is to adapt then
33             MPI_Probe_adapt()        // workers need to know its process status
34             Resource_Adapt()
35         end
36         if instruction is to work then
37             Compute the job              // Compute c_i = f(x_i) and π_posterior(x_i)
38             Write grid point data to output file             // MPI-IO operations
39             Inform Master the job is done
40         end
41     end
42 End
```

The Master Process

The master and worker functions are summarized in Algorithm 7.4. The `Master` process divides the added grid points (resulting from sparse grid creation or refinement) into a number of chunks of equal sizes, and sends each chunk (as a *job*) to a worker. A job array, with each element representing a job, is used for keeping track of job status, i.e., `Todo`, `Progress`, `Done`. The status `Todo` means the job is not yet sent to any worker, `Progress` means the job is sent to a worker but not yet finished, and `Done` means the job is completed. A worker array, with each element representing a worker process, is used for keeping track of the worker status, i.e., either `Active` or `Idle`.

In a while loop, the `Master` process actively checks for active workers for finished jobs, and keeps sending out `Todo` jobs to idle workers. There is no need to collect and aggregate job results, since all computed data are written to files directly by each worker.

The resource adaptation block (line 12–21 in Algorithm 7.4) is executed at a controlled frequency via a counter. Only when there are resource changes and there are more jobs left to do, the `Master` invokes a sequence of operations to interrupt worker computation for a resource adaptation. It first ensures that all workers have finished their current jobs by collecting results from active workers. It then sends out the adapt signal to workers and enters the `Resource_Adapt` block itself. Post adaptation, the `Master` updates the worker array both in size and values to reflect the new resources. To get the workers back in computation, the `Master` must seed them again.

The Worker Processes

The worker function in Algorithm 7.4 is straightforward. A worker waits for the instruction from the `Master` in a while loop and takes actions accordingly. The `Master`'s instruction is encoded in the status object (which is different than the process status) returned by the MPI receive function. The body of the message carries job data if the instruction is to compute, otherwise, it carries dummy values which would be ignored. Upon extraction of the instruction, the worker either computes a job, performs resource adaptation, or terminates. In case of adaptation, the worker must first obtain its process status by calling `MPI_Probe_adapt` before entering the `Resource_Adapt` function.

7.4 Performance Evaluation

In this section, we present the performance and resource efficiency analysis on a statistical inverse problem solver for locating obstacles in a fluid channel. We first conduct tests to determine the impact of resource adaptivity on the application itself, then we compare performance between tests with static and Elastic MPI.

7.4.1 Execution Environment: Virtual Machine Emulated Cluster

Due to limited access to the SuperMUC Petascale System at the time this thesis was written, all tests with this application were conducted on a virtual machine (VM) emulated cluster, which provided an execution environment of 8 nodes with 2 cores per node. Specifications of the cluster as well as its host machine are listed in Table 6.1.

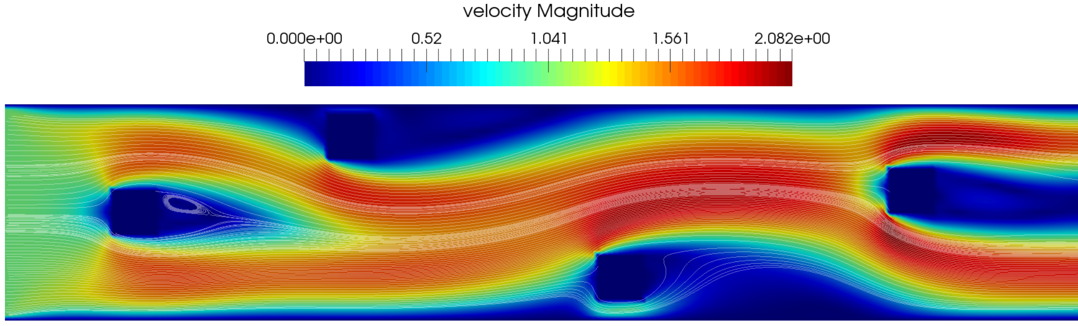


Figure 7.6: A forward simulation of 2-D fluid channel with four obstacles

7.4.2 Benchmark Configuration

The forward simulation with the full model was configured with zero initial values for the pressure and fluid velocities over the entire domain. The inflow velocity on the left boundary was set to 1.0 m/s (meters per second) in the x-direction and 0.0 m/s in the y-direction. External forces in both x- and y-directions were set to zero. Resolution of the domain was restricted to 100×20 grid cells due to the limited computing power of the VM cluster. Figure 7.6 visualizes the fluid velocity of one forward simulation with the obstacles placed at the shown locations.

We assumed 20% Gaussian noise in the observed data, and set

$$\sigma = 0.2 \times \text{mean}(\vec{y}_{\text{observed}}) \tag{7.32}$$

for the posterior computation shown in (7.23).

The surrogate model error, given by

$$e := \frac{1}{Q} \sum_{i=1}^{Q} \frac{\|f_{\text{SGI}}(x_i) - f(x_i)\|}{\|f_{\text{SGI}}(x_i) + f(x_i)\|}, \tag{7.33}$$

was computed with $Q = 50$ randomly chosen input samples. This error was bounded between 0.0 and 1.0, and the surrogate model was refined until $e \leq 0.06$. The final surrogate model varied across tests, because they went through different number of refinement phases due to being tested against different random samples. The surrogate model was set to refine at least once regardless of the model error in order to avoid being overly simplified.

While the *surrogate building* phases were resource-elastic, the *initialization* and the *MCMC solver* phase were static. The parallel tempering MCMC solver was executed on K parallel chains, with K being the number of MPI processes with which the application exited the last surrogate building phase. Excess parallel chains could actually impair accuracy, therefore, K is usually bounded to a certain limit, e.g., 20 in our current setting, but this concern did not apply to the VM cluster due to its small size.

7.4.3 Resource Adaptivity Overhead and Outcome of the Inference

The goal of this experiment was to analyze the impact of resource adaptivity, i.e., the overhead introduced by resource adaptation and the subsequent communication and data movements. We launched every test with a minimal resource assignment, i.e., 2 MPI

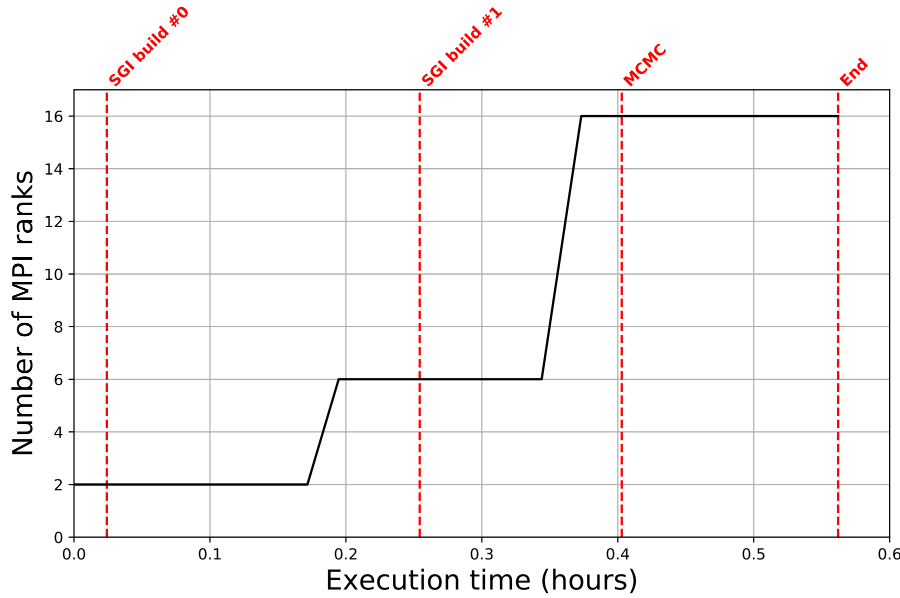


Figure 7.7: Resource profile of one test run of the obstacle location inverse problem

processes (on 1 node). In order to fully examine different cases of resource expansion and reduction, we used a random elastic scheduler that gave a new resource assignment randomly between 1 and 8 nodes every 5 minutes. There was no adaptation if the generated random number was the same as the current number of nodes. The application was set to probe the resource manager every 60 seconds.

Figure 7.7 shows the resource profile of one of these tests. The beginning of each phase is marked by a red vertical line with a tag. The *SGI build* tag denotes a surrogate building or refinement phase. The static initialization phase occurred before the first surrogate building phase, and the static MCMC solver phase occurred after the last surrogate building phase. The *END* tag indicates the end of execution. In this test, there were two surrogate building phases, i.e., the surrogate model was refined once after its initial build. Resource adaptation occurred once during each of these phases (resource adaptation can occur multiple times during a surrogate building phase depending on its duration). The application exited the last surrogate refinement phase with 16 processes. Therefore, the MCMC solver was executed on 16 parallel chains.

Table 7.2 lists the execution time of each computational phase as well as the model error and the computed number of grid points for each surrogate building phase. The total execution time of this test is about 34 minutes, of which the resource-elastic period is about 23 minutes. Table 7.3 provides a summary of the execution time for resource adaptivity. The accumulated time spent on resource adaptation is about 1.2 seconds, which is about 0.06% of the total execution time and 0.09% of the resource-elastic period.

For all tests conducted, the average execution time on Elastic MPI functions and total resource adaptivity were similar to the numbers shown by Table 7.3. The application's total runtime varied significantly across tests due to the fact that it went through different number of phases. With longer runtime, the proportion of resource adaptation overhead

became even less. From this we can conclude that resource adaptivity has negligible impact on runtime for this application.

Table 7.2: Execution time of each phase

Computation phase	Exec. time (sec)	Model error	# Grid points
Initialization	87.03	–	–
SGI build #0	827.06	0.1112	1121
SGI build #1	530.51	0.0384	1306
MCMC solver	572.86	–	–
Total execution	2023.49	–	–

Table 7.3: Execution time of Elastic MPI functions

Elastic MPI function	Avg. time (sec)	Acc. time (sec)	% of total adap.
`MPI_Init_adapt`	0.0513	0.1026	8.50%
`MPI_Probe_adapt`	0.0002	0.0035	0.29%
`MPI_Comm_adapt_begin`	0.2604	0.5208	43.13%
`MPI_Comm_adapt_commit`	0.2107	0.4213	34.89%
Data migration	0.0797	0.1593	13.19%
Total adaptation	0.6038	1.2076	100.00%

Table 7.4: Computational efforts comparison between coupling the MCMC solver with the full simulation model and with surrogate model construction

	With full model	With surrogate model (2427 grid points)
Average simulation time	1.43 (sec)	0.03 (sec)
Surr. construction (CPU hours)	0.00	0.96
MCMC solver 20000 samples (CPU hours)	7.94	0.15
Total computational effort (CPU hours)	7.94	1.11

Outcome of the Inference

The main feature of this inverse problem solver is to reduce computational costs by replacing the full simulation model with a surrogate model. The total computational cost on solving the problem with the full model can be computed by the the simulation time of the full model T_f multiplied by the number of MCMC sampling steps S, i.e.,

$$\text{Cost w. full model} = T_f \times S. \tag{7.34}$$

With surrogate model construction, the total cost is computed by summing the cost of surrogate model construction and the cost of running the MCMC solver with the surrogate model. The cost of surrogate model construction is given by the total number of grid points P in the final adaptive sparse grid multiplied by the simulation time of the full model T_f. Therefore, the total computational cost on solving the problem with the surrogate model

can be computed by

$$\text{Cost w. surr. model} = T_f \times P + T_{f_{\text{SGI}}} \times S. \tag{7.35}$$

Table 7.4 shows the computational cost reduction of the test run shown by Figure 7.7 comparing to a hypothetical MCMC solver coupled with the full simulation model. In this test run, the final sparse grid contained 2427 grid points. And the MCMC solver produced 20000 samples. Based on the calculation given by (7.34) and (7.35), this inverse problem solver reduced the total computational effort from 7.94 to 1.11 CPU hours.

The inference results are visualized by Figure 7.8 and Figure 7.9. Figure 7.8 shows the most probable combination of obstacle locations. Figure 7.9 shows the histogram of the samples produced by the MCMC solver by each dimension. The dashed curve is an auto-computed Gaussian distribution based on the samples. The blue vertical line marks the sample point with the highest posterior value, i.e., the most probable sample point.

7.4.4 Execution Time and Resource Efficiency

In this experiment, we want to compare execution time and resource efficiency of Elastic MPI test runs with the static MPI counterparts. As mentioned, the surrogate model construction process varies across different tests, not due to different parametric settings but due to the randomly chosen test points for the surrogate model evaluation. It is unfair to compare runtime between runs that went through different number of phases, because that means the computational workload in these runs are different, i.e., the more phases a test run has means the more grid points it had computed. Therefore, the two tests we selected for comparison have the same number of computational phases, i.e., each of them went through exactly 2 surrogate building phases.

The surrogate construction process is an embarrassingly parallel problem. Its parallel subtasks are independent of each other and have no requirement for communication or synchronization. Therefore, the application can potentially run on any number of resources. There is not a “fitting” resource assignment profile. For this reason, the Elastic MPI run were produced under a random elastic scheduler.

Figure 7.10 shows the plotting of the resource profiles (number of MPI processes) of the two test runs against execution time in hours. The resource utilization (in CPU hours) of each run is computed by integrating its resource profile curve over its execution time. The CPU hours are visually represented by the shaded area under each curve. The exact execution time and CPU hours of each test are further listed in Table 7.5.

The first run with static MPI at 16 processes completed in about 26 minutes, resulting in 2.86 CPU hours. The second run with Elastic MPI started with 2 processes, adapted twice during the course of execution and completed with 16 processes. It took about 34 minutes to complete, resulting in 2.01 CPU hours. Having the same amount of workload, both runs have comparable resource utilization. It seemed that resource adaptivity did not have a big impact on resource efficiency, and that the application can be flexible with resource assignments.

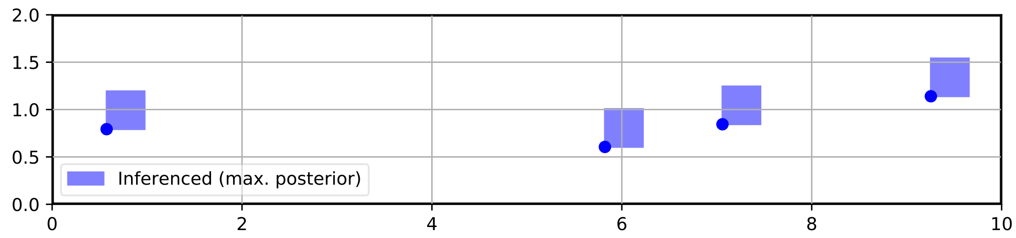


Figure 7.8: Most probable obstacle locations

Figure 7.9: Histogram of the MCMC output samples per dimension

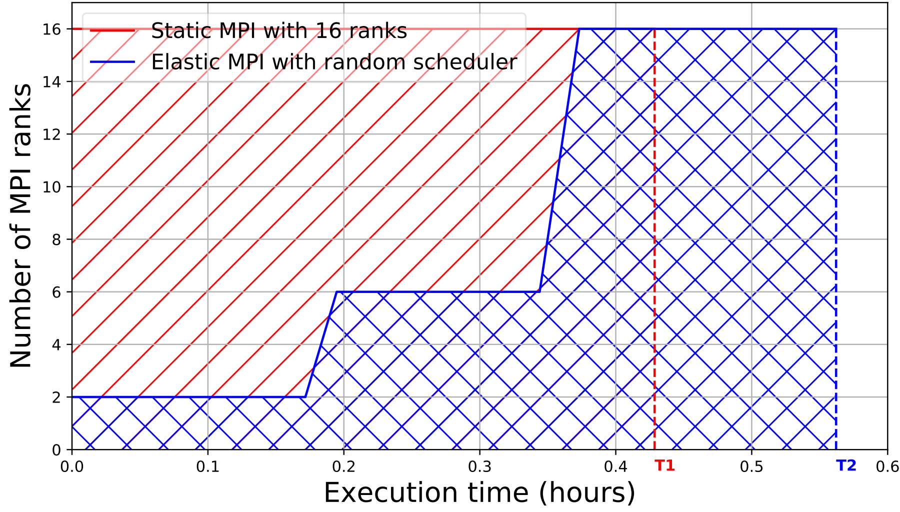


Figure 7.10: Resource profile vs. execution time and CPU hours of two test runs.

Table 7.5: Execution time and CPU hours of two test runs

Test run	Exec. time (hours)	CPU hours
1. Static MPI with 16 processes	0.43	2.86
2. Elastic MPI with random scheduler	0.56	2.01

7.5 Summary

We implemented a statistical inverse problem solver that couples a surrogate model with a parallel MCMC sampler. The solver constructs surrogate models via function interpolation on adaptive sparse grids. This process is designed with resource elasticity, which makes the application go through a number of resource-adaptive computational phases. This solver is generic and can be used on different inverse problems. For our experimentation, we applied it to a problem where we locate obstacles in a fluid channel based on sparse measurements of the fluid velocity. We had to conduct tests for this application on an 8-node VM cluster with 2 CPUs per node due to limited access to SuperMUC.

This solver is an ideal addition to our application base for Elastic MPI investigation, because it is embarrassingly parallel and it implements a master-worker execution model. Rather than having one main compute loop, it goes through multiple resource-elastic phases. The computational workload varies across different phases. However, the stability of workload does not affect its parallel performance due to the nature of the master-worker model.

From the performance analysis, we observe that resource adaptivity introduces negligible overhead and has little impact on the application's execution time and resource efficiency. This is comprehensible, because the application is immune to data redistribution due to the absence of data dependency between parallel subtasks as well as the employment of MPI-IO for data transfer.

In embarrassingly parallel problems, parallel subtasks can be distributed arbitrarily and computed asynchronously. This makes it comparably easy to incorporate resource elasticity from a communication stand point, because changes in resources do not require data redistribution or processes synchronization. We can generalize that embarrassingly parallel applications can act as “buffers” in an elastic environment, in the sense that they can take up unutilized resources in case of resource abundance and that they can release resources to feed other applications in case of a resource shortage.

PART IV

CONCLUSION

8 Conclusion and Outlook

8.1 Conclusion

Motivated to overcome some of the most compelling HPC challenges, we propose a solution that aims to optimize system performance and energy efficiency by realizing resource awareness and elasticity. This approach requires support from both HPC systems and applications. This thesis focuses on the development of resource-aware and elastic applications for distributed-memory HPC systems.

With the Elastic MPI infrastructure and API we developed, we abstracted programming models for different types of parallel applications. We covered Elastic MPI programming models for both SPMD and master-worker execution schemes, which are often implemented for communication-intensive and embarrassingly parallel applications respectively. Moreover, we explained how to handle applications with single and multiple resource-elastic computational phases.

We developed three resource-elastic applications and conducted performance analysis on each individually. Due to their different characteristics, they demonstrated very different behaviors with runtime resource adaptivity. Table 8.1 provides a brief summary on the characteristics of the three applications.

Table 8.1: Characteristics summary of Elastic MPI applications

Application	Communication type	Execution model	Performance bottleneck	Computational workload
Tsunami simulation	communication-intensive	SPMD w. single phase	compute-bound	dynamic
Oil reservoir simulation	communication-intensive	SPMD w. single phase	communication-bound	static
Inverse problem solver w. surrogate construction	embarrassingly parallel	master-worker w. multiple phases	compute-bound	dynamic

Tsunami Simulation

This is a classical communication-intensive grid-based HPC application implementing a SPMD model. It is compute-bound and has a very dynamic computational workload.

Performance tests for this application were conducted in a 32-node environment on the SuperMUC petascale system.

We observed that the overhead introduced by resource adaptation was within an acceptable range, i.e., a few percent of the total execution time. This was mostly attributed to the fact that we leveraged the application's inherent load balancing scheme to avoid performing data redistribution and load balancing during the resource adaptation windows. We also observed that due to the application's dynamic workload, appropriate runtime adjustment on resource allocation helped to improve its resource utilization efficiency with a trade-off on the execution time.

Oil Reservoir Simulation

This is yet another classical communication-intensive grid-based HPC application implementing a SPMD model. This application has a very similar computational workflow to the tsunami simulation. However, it is communication-bound, which means it is more sensitive to communication overhead and it has a mostly stable computational workload. Due to limited access to the SuperMUC petascale system, performance tests for this application were conducted on an 8-node virtual machine emulated cluster.

Results showed that the resource adaptation overhead was insignificant, i.e., less than one percent of the total execution time. This was again due to the fact that we overlapped the application's inherent load balancing operations with data redistribution required by resource adaptation. The application's stable workload was a clear indication that it did not require runtime resource adjustment. Due to its communication-bound characteristics, the application is prone to communication overhead and harder to scale. Test results showed that excess resources could impair performance and that it is important to find a suitable resource assignment. While Elastic MPI was not needed for resource adaptivity for workload, it could help to find and adjust the application to the optimal resource assignment at runtime.

Inverse Problem Solver with Surrogate Construction

This is an embarrassingly parallel application implementing a master-worker model. It has multiple resource-adaptive phases with a different workload for each phase. It is considered compute-bound due to its trivial communication between parallel processes. Performance tests for this application were also performed on an 8-node virtual machine emulated cluster.

Results showed that resource adaptation overhead was insignificant. This was attributed to the application's embarrassingly parallel nature that there is no intensive communication needed for adding or removing resources. Runtime resource adaptation did not improve or decrease the application's resource utilization efficiency. It had, however, an impact on the execution time, i.e., the application could run to completion faster with more resources, and vice versa. Therefore, the application is flexible on resource assignment if there is no requirement for execution time.

Generalization

From the experiments with all three applications, we can generalize the following conclusions:

- Resource adaptation overhead for embarrassingly parallel applications has little impact. For communication-intensive applications with an inherent load balancing scheme, resource adaptation overhead can be reduced significantly by overlapping their built-in load balancing operations with those required for resource adaptation.
- For communication-intensive applications with a dynamic workload, Elastic MPI can help to improve their resource utilization efficiency with appropriate runtime resource assignments that adapt to their changing workload.
- For communication-intensive applications with a static workload, though runtime resource adaptivity for workload is not necessary, Elastic MPI can help to find and adjust the applications to their optimal resource assignments.
- For embarrassingly parallel applications, resource elasticity does not improve or decrease their resource utilization efficiency. When there is no requirement or limitation on execution time, such applications can act as "buffers" in an elastic environment, in the sense that they can take up unutilized resources in case of resource abundance and that they can release resources to feed other applications in need in case of a resource shortage.

In this pilot study on malleable parallel software development with Elastic MPI, we have seen positive impacts of resource awareness and elasticity on individual applications. We are encouraged to continue on this path to achieve improvements on the system level.

8.2 Outlook

In the current Elastic MPI release, there have been known stability issues (discussed in Section 3.4.5) that prevented us from running tests on larger scales. These problems, including those in the fan-out functionality in the PMI layer and those with the internal metadata organization, are mostly rooted in the original design for static resources in the SLURM and MPI bases. The largest Elastic MPI tests on scalability (not presented) we have conducted were on 128 thin nodes and 64 Haswell nodes on SuperMUC. The largest valid runs of the tsunami simulation presented in this dissertation were on 32 thin nodes. Resolving the stability issues is the utmost important task in future Elastic MPI releases. And conducting thorough performance tests on larger scales is one of our next goals for malleable parallel application study.

While the elastic execution support for runtime resource expansion and reduction is fully functional, the elastic resource manager is missing certain crucial functionalities (discussed in Section 3.4.4), which include the batch scheduler and the ability of the runtime scheduler to recognize and handle other execution models besides SPMD. Due to these reasons, we were not able to conduct integration tests, in which multiple elastic applications run concurrently in the same environment to compete for resources. The integration tests can provide performance evaluation on the system level. With future Elastic MPI releases that provide complete resource management functionalities, we can investigate the impacts of resource awareness and elasticity on the overall system performance, throughput and energy efficiency.

In this work, we have experimented with three applications. Further investigation on more applications of different classes and characteristics is necessary. There is a large pop-

ulation of HPC applications that have static structure and workload, and were originally designed for static resources. Such types are important targets for our next investigation.

We will investigate using the mechanisms of elastic resource management for power corridor enforcement. This can help to increase predictability in energy consumption and thus, reduce the overall energy cost.

Elastic MPI can be extended for fault-tolerance. We have the plan to implement a resource-aware checkpointing infrastructure for fast adaptive data recovery.

Bibliography

Remark: The number(s) at the end of each entry indicate the page(s) where the respective reference is cited. In the PDF version, these numbers are hyperlinks.

[1] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive Computing: An Overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, pages 241–268. Springer New York, New York, NY, 2011. `doi:10.1007/978-1-4419-6460-1_11`. 3, 5

[2] M. Snir, editor. *MPI–the complete reference*. Scientific and engineering computation. MIT Press, Cambridge, Mass, 2nd ed edition, 1998. 4

[3] W. Gropp, editor. *The MPI-2 extensions*. Number the complete reference ; Vol. 2 in MPI. MIT Press, Cambridge, Mass., 1998. OCLC: 174807419. 4

[4] MPI Forum. `https://www.mpi-forum.org`, 2018. [Online]. 4, 13

[5] S. Iqbal, R. Gupta, and Y.-C. Fang. Planning considerations for job scheduling in hpc clusters. *Dell Power Solutions*, pages 133–136, 2005. 4

[6] Transregional Research Center InvasIC. `http://www.invasic.de`, 2018. [Online]. 5

[7] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, and J. Weidendorfer. Invasive computing with iomp. In *Specification and Design Languages (FDL), 2012 Forum on*, pages 225–231. IEEE, 2012. 7

[8] A. Hollmann and M. Gerndt. Invasive computing: An application assisted resource management approach. In V. Pankratius and M. Philippsen, editors, *Multicore Software Engineering, Performance, and Tools*, pages 82–85, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 7

[9] M. Schreiber, H.-J. Bungartz, and M. Bader. Shared memory parallelization of fully-adaptive simulations using a dynamic tree-split and-join approach. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, 2012. 7

[10] M. Schreiber. *Cluster-Based Parallelization of Simulations on Dynamically Adaptive Grids and Dynamic Resource Management.* Doctoral dissertation, Technical University of Munich, Munich, Germany, 2014. 7

[11] M. Schreiber, C. Riesinger, T. Neckel, H.-J. Bungartz, and A. Breuer. Invasive compute balancing for applications with shared and hybrid parallelization. *International Journal of Parallel Programming*, 43(6):1004–1027, Dec 2015. doi:10.1007/s10766-014-0336-3. 7

[12] I. Comprés. *Resource-Elasticity Support for Distributed Memory HPC Applications*. Doctoral dissertation, Technical University of Munich, Munich, Germany, 2017. 7, 23, 36, 68

[13] SLURM Workload Manager. https://slurm.schedmd.com, 2018. [Online]. 7, 30

[14] MPICH: High-Performance Portable MPI. https://www.mpich.org, 2018. [Online]. 7, 11, 25

[15] Open MPI: Open Source High Performance Computing. https://www.open-mpi.de, 2018. [Online]. 11

[16] Intel MPI Library. https://software.intel.com/en-us/intel-mpi-library, 2018. [Online]. 11

[17] IBM Spectrum MPI. https://www.ibm.com/us-en/marketplace/spectrum-mpi, 2018. [Online]. 11

[18] D. Holmes, K. Mohror, R. E. Grant, A. Skjellum, M. Schulz, W. Bland, and J. M. Squyres. Mpi sessions: Leveraging runtime infrastructure to increase scalability of applications at exascale. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 121–129, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2966884.2966915, doi:10.1145/2966884.2966915. 13

[19] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, August 2013. URL: http://journals.sagepub.com/doi/10.1177/1094342013488238, doi:10.1177/1094342013488238. 13

[20] Fault Tolerance Research Hub. http://fault-tolerance.org, 2018. [Online]. 13

[21] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906, New Orleans, LA, USA, November 2014. IEEE. URL: http://ieeexplore.ieee.org/document/7013060/, doi:10.1109/SC.2014.78. 13

[22] A. Hassani, A. Skjellum, and R. Brightwell. Design and Evaluation of FA-MPI, a Transactional Resilience Scheme for Non-blocking MPI. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 750–755, Atlanta, GA, USA, June 2014. IEEE. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6903636, doi:10.1109/DSN.2014.78. 13

[23] A. Hassani, A. Skjellum, P. V. Bangalore, and R. Brightwell. Practical resilient cases for fa-mpi, a transactional fault-tolerant mpi. In *Proceedings of the 3rd Workshop on Exascale MPI*, ExaMPI '15, pages 1:1–1:10, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2831129.2831130`, `doi:10.1145/2831129.2831130`. 13

[24] I. Laguna, T. Gamblin, K. Mohror, M. Schulz, H. Pritchard, and N. Davis. A global exception fault tolerance model for mpi. 9 2014. 13

[25] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, and H. Subramoni. ER einit : Scalable and efficient fault-tolerance for bulk-synchronous MPI applications: ER einit : Scalable and efficient fault-tolerance for bulk-synchronous MPI applications. *Concurrency and Computation: Practice and Experience*, page e4863, August 2018. URL: `http://doi.wiley.com/10.1002/cpe.4863`, `doi:10.1002/cpe.4863`. 13

[26] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. d. Supinski, N. Maruyama, and S. Matsuoka. FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1225–1234, Phoenix, AZ, USA, May 2014. IEEE. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6877350`, `doi:10.1109/IPDPS.2014.126`. 13

[27] Charm++: Parallel Programming with Migratable Objects. `http://charm.cs.illinois.edu/research/charm`, 2018. [Online]. 14

[28] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, October 1993. `doi:10.1145/167962.165874`. 14

[29] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. pages 647–658. IEEE, November 2014. `doi:10.1109/SC.2014.58`. 14

[30] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale. Charm++ and MPI: Combining the Best of Both Worlds. pages 655–664. IEEE, May 2015. `doi:10.1109/IPDPS.2015.102`. 14

[31] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive mpi. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, pages 306–322, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 14

[32] C. Huang, G. Zheng, L. Kalé, and S. Kumar. Performance evaluation of adaptive MPI. page 12. ACM Press, 2006. `doi:10.1145/1122971.1122976`. 14

[33] C. Huang, G. Zheng, and L. V. Kalé. Supporting adaptivity in mpi for dynamic parallel applications. *Rapport technique, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign*, 2007. 14

[34] L. Kale, S. Kumar, and J. DeSouza. A Malleable-Job System for Timeshared Parallel Machines. pages 230–230. IEEE, 2002. doi:10.1109/CCGRID.2002.1017131. 14

[35] A. Gupta, B. Acun, O. Sarood, and L. V. Kale. Towards realizing the potential of malleable jobs. pages 1–10. IEEE, December 2014. doi:10.1109/HiPC.2014.7116905. 14

[36] HPX. http://stellar.cct.lsu.edu/projects/hpx/, 2018. [Online]. 15

[37] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014. 15

[38] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser. HPX – An open source C++ Standard Library for Parallelism and Concurrency. In *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017 (OpenSuCo'17)*, page 5, 2017. 15

[39] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009. 15

[40] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. Preliminary design examination of the parallex system from a software and hardware perspective. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):81–87, 2011. 15

[41] M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. An application driven analysis of the parallex execution model, 2011. arXiv:arXiv:1109.5201. 15

[42] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519, October 2005. doi:10.1145/1103845.1094852. 16

[43] V. Saraswat. X10: Concurrent programming for modern architectures. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS'07, pages 1–1, Berlin, Heidelberg, 2007. Springer-Verlag. 16

[44] P. Murthy. Parallel computing with x10. page 5. ACM Press, 2008. doi:10.1145/1370082.1370086. 16

[45] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau. Resource-aware programming and simulation of MPSoC architectures through extension of X10. page 48. ACM Press, 2011. doi:10.1145/1988932.1988941. 16

[46] A. Zwinkau. Resource awareness for efficiency in high-level programming languages. Technical Report 12, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2011. 16

[47] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau. An X10 Compiler for Invasive Architectures. Technical report, Karlsruhe, 2012. doi:10.5445/IR/1000028112. 16

[48] T. Desell, K. E. Maghraoui, and C. A. Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, August 2007. `doi:10.1007/s10586-007-0032-9`. 17

[49] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. The Internet Operating System: Middleware for Adaptive Distributed Computing. *The International Journal of High Performance Computing Applications*, 20(4):467–480, November 2006. `doi:10.1177/1094342006068411`. 17

[50] SALSA Programming Language. `http://wcl.cs.rpi.edu/salsa/`, 2018. [Online]. 17

[51] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIG\-PLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. 17

[52] K. El Maghraoui, B. K. Szymanski, and C. Varela. An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3911, pages 258–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. `doi:10.1007/11752578_32`. 17

[53] PVM Parallel Virtual Machine. `https://www.csm.ornl.gov/pvm/`, 2018. [Online]. 17

[54] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. PVM and HeNCE: Tools for heterogeneous network computing. In *Software for Parallel Computation*, pages 91–99. Springer, 1993. 17

[55] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency and Computation: Practice and Experience*, 2(4):315–339, 1990. 17

[56] T. G. Mattson. Programming environments for parallel and distributed computing: A comparison of P4, PVM, Linda, and TCGMSG. *The International journal of supercomputer applications and high performance computing*, 9(2):138–161, 1995. 17

[57] O. Sonmez, H. Mohamed, W. Lammers, D. Epema, et al. Scheduling malleable applications in multicluster systems. In *Cluster Computing, 2007 IEEE International Conference on*, pages 372–381. IEEE, 2007. 18

[58] N. Beigi-Mohammadi and M. Litoiu. Engineering Self-Adaptive Applications on Cloud with Software Defined Networks. pages 9–12. IEEE, April 2017. `doi:10.1109/IC2E.2017.43`. 18

[59] P. Zoghi, M. Shtern, M. Litoiu, and H. Ghanbari. Designing adaptive applications deployed on cloud environments. *ACM Trans. Auton. Adapt. Syst.*, 10(4):25:1–25:26, January 2016. `doi:10.1145/2822896`. 18

[60] A. Bhadani and S. Chaudhary. Performance evaluation of web servers using central load balancing policy over virtual machines on cloud. In *Proceedings of the Third Annual ACM Bangalore Conference*, page 16. ACM, 2010. 18

[61] S. G. Domanal and G. R. M. Reddy. Load balancing in cloud computingusing modified throttled algorithm. In *Cloud Computing in Emerging Markets (CCEM), 2013 IEEE International Conference on*, pages 1–5. IEEE, 2013. 18

[62] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 18

[63] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010. 18

[64] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *AcM sIGMoD Record*, 40(4):11–20, 2012. 18

[65] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, and H.-J. Bungartz. Infrastructure and API Extensions for Elastic Execution of MPI Applications. pages 82–97. ACM Press, 2016. `doi:10.1145/2966884.2966917`. 23, 68

[66] K. Fürlinger and D. Skinner. Capturing and visualizing event flow graphs of mpi applications. In *European Conference on Parallel Processing*, pages 218–227. Springer, 2009. 34

[67] X. Aguilar, K. Fürlinger, and E. Laure. Mpi trace compression using event flow graphs. In *European Conference on Parallel Processing*, pages 1–12. Springer, 2014. 34

[68] X. Aguilar, K. Fürlinger, and E. Laure. Automatic on-line detection of mpi application structure with event flow graphs. In *European Conference on Parallel Processing*, pages 70–81. Springer, 2015. 34

[69] I. Lee, C. S. Iliopoulos, and K. Park. Linear time algorithm for the longest common repeat problem. *Journal of Discrete Algorithms*, 5(2):243–249, 2007. 35

[70] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. 35

[71] R. Tarjan. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107. ACM, 1973. 35

[72] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997. 35

[73] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, 1999. 35

[74] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *International Static Analysis Symposium*, pages 170–183. Springer, 2007. 35

[75] Leibniz supercomputing centre of the bavarian academy of sciences and humanities. `https://www.lrz.de/services/compute/supermuc`, December 2018. [Online]. 38, 67

[76] B. Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010. 39

[77] O. Meister. *Sierpinski Curves for Parallel AdaptiveMeshRefinement in Finite Element and Finite Volume Methods*. Doctoral dissertation, Technical University of Munich, Munich, Germany, 2016. 55, 58, 59, 64, 72, 75, 77, 78

[78] D. F. Wallace. *Everything and more: a compact history of infinity*. Great discoveries. Atlas Book, New York, 1st ed edition, 2003. 56

[79] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, Mar 1890. `doi:10.1007/BF01199438`. 56

[80] D. Hubert. Über die stetige abbildung einer linie auf ein flächenstück. *Mathematische Annalen*, 38:459–460, 1891. 56

[81] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966. 56

[82] H. Sagan. *Space filling curves*. Universitext. Springer, New York Berlin, 1994. OCLC: 246915680. 56

[83] M. Bader. *Space-filling curves: an introduction with applications in scientific computing*. Number 9 in Texts in computational science and engineering. Springer, Heidelberg ; New York, 2013. 56

[84] W. Sierpiński. Sur une nouvelle courbe continue qui remplit toute une aire plane. *Bull. Acad. Sci. Cracovie (Sci. math. et nat. Serie A)*, pages 462–478, 1912. 56

[85] W. F. Mitchell. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *Journal of Computational and Applied Mathematics*, 36(1):65–78, August 1991. `doi:10.1016/0377-0427(91)90226-A`. 56

[86] W. F. Mitchell. 30 years of newest vertex bisection. page 020011, 2016. `doi:10.1063/1.4951755`. 56

[87] M. Bader, S. Schraufstetter, C. Vigh, and J. Behrens. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves. *International Journal of Computational Science and Engineering*, 4(1):12, 2008. `doi:10.1504/IJCSE.2008.021108`. 56

[88] M. Bader and C. Zenger. Cache oblivious matrix multiplication using an element ordering based on a Peano curve. *Linear Algebra and its Applications*, 417(2-3):301–313, September 2006. `doi:10.1016/j.laa.2006.03.018`. 57

[89] M. Bader, K. Rahnema, and C. Vigh. Memory-Efficient Sierpinski-Order Traversals on Dynamically Adaptive, Recursively Structured Triangular Grids. In D. Hutchison,

T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and K. Jónasson, editors, *Applied Parallel and Scientific Computing*, volume 7134, pages 302–312. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 57

[90] R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge texts in applied mathematics. Cambridge University Press, Cambridge ; New York, 2002. 58, 80

[91] D. S. Bale, R. J. LeVeque, S. Mitran, and J. A. Rossmanith. A Wave Propagation Method for Conservation Laws and Balance Laws with Spatially Varying Flux Functions. *SIAM Journal on Scientific Computing*, 24(3):955–978, jan 2003. doi:10.1137/S106482750139738X. 59

[92] T. Gallouët, J.-M. Hérard, and N. Seguin. Some approximate Godunov schemes to compute shallow-water equations with topography. *Computers & Fluids*, 32(4):479 – 513, 2003. doi:10.1016/S0045-7930(02)00011-7. 59

[93] D. L. George. Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. *Journal of Computational Physics*, 227(6):3089 – 3113, 2008. doi:10.1016/j.jcp.2007.10.027. 59

[94] E. Godlewski and P.-A. Raviart. *Numerical Approximation of Hyperbolic Systems of Conservation Laws*. Springer, New York, NY, 1996. OCLC: 879624121. 59

[95] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, May 2011. doi:10.1017/S0962492911000043. 59

[96] G. Zumbusch. On the quality of space-filling curve induced partitions. *Z. Angew. Math. Mech.*, 81:25–28, 2001. Suppl. 1, also as report SFB 256, University Bonn, no.674, 2000. 60

[97] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976. doi:10.1016/0304-3975(76)90059-1. 60

[98] R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of np-completeness. *WH Free. Co., San Fr*, pages 90–91, 1979. 62

[99] A. Pınar and C. Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004. 62

[100] GEBCO_2014 Grid. http://www.gebco.net/data_and_products/gridded_bathymetry_data/gebco_30_second_grid, 2016. [Online]. 66

[101] SPE10 – Society of Petroleum Engineers comparative solution project 10 model 2. http://www.spe.org/web/csp/datasets/set02.htm, 2018. [Online]. 76

[102] O. Meister and M. Bader. 2d adaptivity for 3d problems: Parallel SPE10 reservoir simulation on dynamically adaptive prism grids. *Journal of Computational Science*, 9:101–106, July 2015. doi:10.1016/j.jocs.2015.04.016. 78

[103] R. Helmig, J. Niessner, B. Flemisch, M. Wolff, and J. Fritz. Efficient modeling of flow and transport in porous media using multiphysics and multiscale approaches. In *Handbook of geomathematics*, pages 417–457. Springer, 2010. 79

[104] R. Temam. *Navier-Stokes equations: theory and numerical analysis.* AMS Chelsea Pub, Providence, R.I, 2001. 79, 112

[105] A. Majda and A. L. Bertozzi. *Vorticity and incompressible flow.* Cambridge texts in applied mathematics. Cambridge University Press, Cambridge ; New York, 2002. 79, 112

[106] J. E. Aarnes, V. Kippe, and K.-A. Lie. Mixed multiscale finite elements and streamline methods for reservoir simulation of large geomodels. *Advances in Water Resources*, 28(3):257–271, March 2005. URL: http://linkinghub.elsevier.com/retrieve/pii/S0309170804001885, doi:10.1016/j.advwatres.2004.10.007. 81

[107] M. Christie and M. Blunt. Tenth SPE Comparative Solution Project: A Comparison of Upscaling Techniques. *SPE Reservoir Evaluation & Engineering*, 4(04):308–317, August 2001. URL: http://www.onepetro.org/doi/10.2118/72469-PA, doi:10.2118/72469-PA. 81

[108] M. Wolff. *Multi-scale modeling of two-phase flow in porous media including capillary pressure effects.* Doctoral dissertation, Universität Stuttgart, Stuttgart, Germany, 2013. doi:10.18419/opus-501. 81

[109] K. Aziz and A. Settari. *Petroleum reservoir simulation.* Applied Science Publishers, London, 1979. 81

[110] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100(1):32–74, December 1928. URL: https://doi.org/10.1007/BF01448839, doi:10.1007/BF01448839. 83

[111] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM J. Res. Dev.*, 11(2):215–234, March 1967. doi:10.1147/rd.112.0215. 83

[112] L. T. Biegler, editor. *Large-scale inverse problems and quantification of uncertainty.* Wiley series in computational statistics. Wiley, Chichester, West Sussex, 2011. 100

[113] J. Kaipio and E. Somersalo. *Statistical and computational inverse problems.* Number v. 160 in Applied mathematical sciences. Springer, New York, 2005. 100, 101, 102

[114] H. Jeffreys et al. *Scientific inference.* Cambridge University Press, 1973. 101

[115] R. Bellman. *Dynamic programming.* Dover Publications, Mineola, N.Y, dover ed edition, 2003. 102

[116] F. Liang, C. Liu, and R. J. Carroll. *Advanced Markov chain Monte Carlo methods: learning from past samples.* Wiley series in computational statistics. Wiley, Chichester, West Sussex, U.K, 2010. 102

[117] S. Brooks, editor. *Handbook for Markov chain Monte Carlo.* Taylor & Francis, Boca Raton, 2011. 102

[118] D. Gamerman and H. F. Lopes. *Markov chain Monte Carlo: stochastic simulation for Bayesian inference.* Number 68 in Texts in statistical science series. Taylor & Francis, Boca Raton, 2nd ed edition, 2006. 102

[119] C. M. Bishop. *Pattern recognition and machine learning.* Information science and statistics. Springer, New York, 2006. 102

[120] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta numerica*, 13:147–269, 2004. 107, 108, 109, 110

[121] M. Griebel. *Sparse grids and related approximation schemes for higher dimensional problems.* Citeseer, 2005. 107, 108, 109, 110

[122] D. Pflüger. *Spatially adaptive sparse grids for high-dimensional problems.* Verl. Dr. Hut, München, 1. aufl edition, 2010. OCLC: 700066168. 107, 108, 109, 110

[123] J. Garcke. Sparse grids in a nutshell. In *Sparse grids and applications*, pages 57–80. Springer, 2012. 107, 108, 109, 110

[124] T. Gerstner and M. Griebel. Numerical integration using sparse grids. *Numerical algorithms*, 18(3-4):209, 1998. 107

[125] V. Barthelmann, E. Novak, and K. Ritter. High dimensional polynomial interpolation on sparse grids. *Advances in Computational Mathematics*, 12(4):273–288, 2000. 107, 109

[126] J. Garcke, M. Griebel, and M. Thess. Data mining with sparse grids. *Computing*, 67(3):225–253, 2001. 107

[127] M. Hegland. Adaptive sparse grids. *Anziam Journal*, 44:335–353, 2003. 107

[128] J. D. Jakeman and S. G. Roberts. Local and dimension adaptive sparse grid interpolation and quadrature. *CoRR*, abs/1110.0010, 2011. 109

[129] P. Neumann, A. Atanasov, and C. Kowitz. Praktikum wissenschaftliches rechnen computational fluid dynamics, 2012. 112, 114

[130] P. J. Olver. *Introduction to partial differential equations.* Springer Science+Business Media, LLC, New York, NY, 2013. 113

[131] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 2007. OCLC: ocm86110147. 113

[132] J. C. Butcher. *Numerical methods for ordinary differential equations.* Wiley, Chichester, England ; Hoboken, NJ, 2nd ed edition, 2008. OCLC: ocn191024153. 113

[133] A. D. Polyanin. *Handbook of linear partial differential equations for engineers and scientists.* Chapman & Hall/CRC, Boca Raton, 2002. 114

[134] D. Medková. *The Laplace Equation: Boundary Value Problems on Bounded and Unbounded Lipschitz Domains.* Springer International Publishing Springer, Cham, 2018. 114