A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:

http://wrap.warwick.ac.uk/186567

Copyright and reuse:
This thesis is made available online and is protected by original copyright. Please scroll down to view the document itself. Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk
Predictive Analysis and Optimization of Large-Scale Coupled Multi-Physics Simulations Using Mini-Apps

Archie William Powell

A thesis submitted to The University of Warwick in partial fulfillment of the requirements for admission to the degree of

Doctor of Philosophy in Computer Science

September 2023
# Contents

List of Figures v

List of Tables x

Acknowledgments xi

Declarations xii

Abstract xiii

Abbreviations xiv

1 Introduction 1

1.1 Motivation and Problem Statement 3

1.2 Thesis Contributions 6

1.3 Thesis Limitations 8

1.4 Thesis Overview 9

2 Background and Related Work 11

2.1 Code coupling 11

2.2 Coupling in multi-physics problems 15

2.2.1 Fluid-Structure Interaction 15

2.2.2 Fission Reactor Fuel 15

2.2.3 Magnetic Confinement Fusion 16

2.2.4 Climate Modeling 16

2.3 Coupling in a gas-turbine aero engine 17
## Contents

2.3.1 Coupling in a gas-turbine compressor .................................................. 17  
2.3.2 Coupling in a full gas-turbine engine .................................................. 20  
2.3.3 Design of the Rolls-Royce (RR) Coupler ........................................... 22  

2.4 Mini-apps and a mini-simulation .............................................................. 28  
2.4.1 Mini-apps ............................................................................................... 28  
2.4.2 Building a mini-simulation ..................................................................... 29  
2.4.3 MG-CFD - A density solver proxy ......................................................... 29  
2.4.4 SIMPIC - A pressure solver proxy ......................................................... 30  
2.4.5 Mini-simulation setup .............................................................................. 32  

2.5 Performance modeling .............................................................................. 33  

2.6 Summary ..................................................................................................... 34  

3 Building a Mini-Coupler and Compressor Mini-Simulation ....................... 36  
3.1 Building the CPX mini-coupler ................................................................. 36  
3.1.1 Initial design considerations of a mini-coupler ..................................... 36  
3.1.2 Components of the Rolls-Royce (RR) Coupler and design of CPX ........ 38  
3.1.3 Capturing performance behavior ............................................................. 44  
3.2 Building the empirical performance model ............................................... 45  
3.2.1 Initial run-time ....................................................................................... 47  
3.2.2 Fitting application parallel efficiency .................................................... 50  
3.2.3 Deriving run-time from parallel efficiency ............................................ 55  
3.3 Applying the empirical performance model ............................................. 56  
3.3.1 Iteration example .................................................................................. 58  
3.3.2 Algorithm inputs and outputs ............................................................... 59  
3.4 Summary of performance model design ............................................... 60  
3.5 Building the mini-simulation ................................................................. 61  
3.5.1 Production-grade problem ................................................................. 61  
3.5.2 Mini-app equivalent ............................................................................... 62  
3.6 Summary ..................................................................................................... 62  

4 Performance Analysis and Prediction of Compressor Test Cases ............... 63  
4.1 Comparison with an internal RR test case .............................................. 64  

CONTENTS

4.2 Comparison with Rig250 coarse subset . . . . . . . . . . . . . . . . . . . . 67
4.3 Validating the Performance model - Rig250 coarse subset . . . . . . . . 70
4.4 Predicting the performance of the full Rig250 . . . . . . . . . . . . . . . 71
  4.4.1 Base performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
  4.4.2 Optimized coupler . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
  4.4.3 Updated performance model with Rig250 coarse subset . . . . . . . 75
  4.4.4 Updated performance model with full Rig250 . . . . . . . . . . . . . 77
4.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

5 Building and Optimizing a Full Engine Simulation 81
  5.1 Extending CPX for a full-engine mini-simulation . . . . . . . . . . . . . . 84
    5.1.1 SIMPIC integration . . . . . . . . . . . . . . . . . . . . . . . . . . 84
    5.1.2 Solver input . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
    5.1.3 Other changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
  5.2 Configuration of the full-engine simulation . . . . . . . . . . . . . . . . . 89
    5.2.1 Combustion test cases . . . . . . . . . . . . . . . . . . . . . . . . . 89
    5.2.2 Setup and challenges of the full-engine simulation . . . . . . . . . . 90
  5.3 Building the performance proxy and identifying the simulation bottleneck 91
    5.3.1 Simulation bottleneck . . . . . . . . . . . . . . . . . . . . . . . . . 93
  5.4 Further performance analysis and optimization of the Pressure Solver . . 95
    5.4.1 Spray methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
    5.4.2 Pressure field . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
    5.4.3 Extrapolating improvements . . . . . . . . . . . . . . . . . . . . . 100
  5.5 Extending the empirical performance model . . . . . . . . . . . . . . . . . 102
    5.5.1 Initial run-time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
    5.5.2 Modeling SIMPIC parallel efficiency . . . . . . . . . . . . . . . . . . 104
    5.5.3 Other changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
  5.6 Applying the extended performance model . . . . . . . . . . . . . . . . . . 105
    5.6.1 Building the iterative algorithm . . . . . . . . . . . . . . . . . . . . 105
    5.6.2 Iteration example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
    5.6.3 Comparison to original algorithm . . . . . . . . . . . . . . . . . . . . 108
  5.7 Evaluating the full engine test cases . . . . . . . . . . . . . . . . . . . . . 108
5.8 Comparison with a small 3-stage test case . . . . . . . . . . . . . . . . . . . 109
5.9 Comparison with the 16-stage full-engine test case . . . . . . . . . . . . . 111
5.10 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

6 Conclusions and Future Work 116

6.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
  6.2.1 Improving CPX and exploring different coupling methods . . . . . . 120
  6.2.2 Improving the accuracy of the performance model . . . . . . . . . . . 121
  6.2.3 Further validation and extended simulations . . . . . . . . . . . . . . 122
6.3 Final thoughts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
List of Figures

1.1 RR Trent XWB Engine [Rolls-Royce, Accessed May 2022] (Reproduced with Permission) ................................................................. 3

1.2 An annotated model of the DLR Rig250 compressor test case (top) and setup of a Rotor/Stator test case with Density Solver instances (DSs) and Coupler Units (CUs) (bottom) [Marciniak et al., 2014] (Reproduced with Permission) ................................................................. 5

2.1 Interpolation between coupled interfaces. The colors represent the different values in the interfaces, which after data exchange and interpolation then match. ................................................................. 12

2.2 A comparison highlighting the issues of peer-to-peer coupling when the coupling routines are computationally heavy ................................................................. 13

2.3 Using a distributed client server-coupling model, the run-time can be reduced if the coupling work is computationally heavy ................................................................. 14

2.4 With mixing plane coupling, the rotor/stator interaction is treated as a steady-state problem and the rotor does not move relative to the stator. ................................................................. 18

2.5 With sliding plane coupling, the rotor is moving relative to the stator requiring the mapping between the two boundaries to be recomputed every time-step. ................................................................. 18

2.6 The three core stages of a simulation using the Rolls-Royce (RR) Coupler [Amirante et al., 2021b] ................................................................. 23

2.7 In a brute force search, the number of memory accesses per interface is equal to the square of the size. ................................................................. 24
2.8 Allocating multiple ranks per coupler unit divides up the search, but each cell must search the entire opposing boundary (a). By manually partitioning the interface, the search space on the opposite boundary can be restricted (b). Here, the interface search routines have been split across 4 Message Passing Interface (MPI) ranks in total, shown by the different colors in each interface. .............................................................. 26

2.9 A comparison between the compute-communication pattern of both the pressure-solver and SIMPIC [Houzeaux et al., 2016] [Thari et al., 2021] [SIMPIC, Accessed May 2022] ................................................................. 31

2.10 A breakdown of the 1.5Bn Rolls-Royce test case setup for both the mini-app and full-scale simulation (Right). The chambers have been labeled on a representative RR Trent XWB Engine [Rolls-Royce, Accessed May 2022] (Left, reproduced with permission). ..................................................... 32

3.1 A typical CPX input configuration file (left), specifying three MG-CFD Sessions, each with 200 Message Passing Interface (MPI) ranks, and two Coupler Units, each with 20 Message Passing Interface (MPI) ranks. The generated configuration (right), with Multi-Unit Mode (MUM) enabled. .. 38

3.2 The two structures which contain coupling information in CPX ............ 39

3.3 The Message Passing Interface (MPI) Commworld communicator must be split up into smaller sub-communicators (shown in blue) for each solver or coupler unit instance to handle their internal communication. ............... 40

3.4 The CPX Coupler Unit (CU) algorithm in comparison to the RR Coupler Coupler Unit (CU) algorithm. CPX uses linear interpolation, a simplified version of the interpolation scheme used in the RR Coupler [Amirante et al., 2021b]. ................................................................. 42

3.5 The model must first generate an initial run-time for the user’s density solver test case. ............................................................. 48

3.6 The model generates an initial run-time for the user’s density solver test case as well as the coupling time .............................. 50

3.7 The effect on parallel efficiency when the amount of serial code in a parallel application increases. ..................................................... 51
LIST OF FIGURES

3.8 The error of modelling in a highly serial-bound application to 0% P.E, seen in a), with the reduction in error by limiting modelling to 50% P.E shown in b). .......................................................... 52
3.9 An example of a loop iteration of the resource allocation algorithm. .... 59
3.10 A breakdown of how the empirical model is used to generate the resource allocation and predict the run-time of a coupled simulation. ............. 60
3.11 An annotated model of the DLR Rig250 compressor test case (top) and setup of a Rotor/Stator test case with Density Solver instances (DSs) and Coupler Units (CUs) (bottom) [Marciniak et al., 2014] (Reproduced with Permission) ..................................................... 61
4.1 A comparison between the three tested Coupler Unit configurations in the industry simulation and the mini-app simulation running this internal RR test case. ................................................. 64
4.2 Rolls-Royce Coupler vs. CPX. 1st-10M cell test case with 1 Coupler Unit, 4 Coupler Units and 1 Coupler Unit with 4 Message Passing Interface (MPI) ranks. .................................................. 66
4.3 A comparison between the three tested Coupler Unit configurations in the industry simulation and the mini-app simulation running the industry motivated two-row Rig250 coarse subset problem. ............... 67
4.4 Rolls-Royce Coupler vs. CPX. 2nd-10M cell test case with 1, 2, and 4 Coupler Units. .................................................. 68
4.5 Rolls-Royce Coupler vs. CPX vs. Model Rig250 coarse subset, testing different CU numbers with 400 Density Solver/MG-CFD Message Passing Interface (MPI) ranks. .................................................. 70
4.6 The predicted run-times (s) for the 4.6B cell test case on Central Processing Unit (CPU) and Graphics Processing Unit (GPU) clusters. The number of Density Solver ranks are fixed at 3700 per Density Solver (DS) for the Central Processing Unit (CPU) cluster and 40 Graphics Processing Unit (GPU)s per Density Solver (DS) for the Graphics Processing Unit (GPU) cluster. .................................................. 72
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>The predicted run-times(s) for the 4.6B cell test on Central Processing Unit</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>(CPU) and Graphics Processing Unit (GPU) power equivalent clusters.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>A comparison between the empirical model and industry simulation in the</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Rig250 Coarse Subset mesh.</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>A comparison between the empirical model and industry simulation in the</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>Rig250 full annulus test case.</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>A comparison between the predicted coupling time of the empirical model</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>and the coupling time of the industry simulation in the Rig250 full annulus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>test case.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>A breakdown of the 1.5Bn Rolls-Royce test case setup for both the mini-app</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>and full-scale simulation (Right). The chambers have been labeled on a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>representative RR Trent XWB Engine [Rolls-Royce, Accessed May 2022] (Left,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reproduced with permission).</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>The CPX input configuration file seen in Figure 3.1 (left), specifying three</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>MG-CFD Sessions, each with 200 Message Passing Interface (MPI) ranks, and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>two Coupler Units, each with 20 Message Passing Interface (MPI) ranks.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>To the right is an updated CPX input file with the final unit switched for a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIMPIC instance, along with a declaration of coupling type between each</td>
<td></td>
</tr>
<tr>
<td></td>
<td>solver instance.</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>The SIMPIC input parameters of a typical test case.</td>
<td>87</td>
</tr>
<tr>
<td>5.4</td>
<td>An example mg_files.input file, specifying the input file for the 1st, 2nd</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>and 4th units (each MG-CFD) and a NULL for the 3rd unit (SIMPIC).</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>A breakdown of the 1.5Bn Rolls-Royce test case setup for both the mini-app</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>and full-scale simulation.</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Parallel efficiency of the pressure solver and SIMPIC on ARCHER2.</td>
<td>92</td>
</tr>
<tr>
<td>5.7</td>
<td>Speedup of the pressure solver and SIMPIC on ARCHER2.</td>
<td>93</td>
</tr>
<tr>
<td>5.8</td>
<td>The speed-up of SIMPIC with the representative large base test case on</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>ARCHER2.</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>A breakdown of the most time-consuming pressure solver functions, exam-</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>ining run-time as a proportion of total run-time at 2048 ARCHER2 cores in a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28M cell test case.</td>
<td></td>
</tr>
</tbody>
</table>
5.10 A breakdown of the most time-consuming pressure solver functions, examining parallel efficiency of each function from 128 to 2048 ARCHER2 cores in a 28M cell test case. ................................................. 97
5.11 Predicted pressure solver parallel efficiency before and after particle and solver optimizations. ......................................................... 100
5.12 Predicted parallel efficiency of an optimized pressure solver and the actual parallel efficiency of the Optimized SIMPIC Test Case (Optimized-STC) on ARCHER2 ................................................. 101
5.13 Predicted speedup of an optimized pressure solver and the actual speedup of the Optimized SIMPIC Test Case (Optimized-STC) on ARCHER2 . . . . . 102
5.14 An example of a loop iteration of the extended resource allocation algorithm.107
5.15 A comparison between predicted and actual run-times of MG-CFD and SIMPIC in a small 3-stage test case. .............................. 110
5.16 The percentage error between the mini-apps and predictive model for individual mini-app simulations using the Base-STC and the Optimized-STC. 112
5.17 The predicted and actual speedups for 1 revolution of the mini-app simulation using the Optimized-STC compared to a mini-app simulation using the Base-STC. ......................................................... 113
List of Tables

3.1 Converting from production to mini-app cycles .......................... 46
3.2 The model must store the size of the mesh used in the simulation .... 47
3.3 The model must store the size of the interface used in the simulation,
calculated from the mesh size and the fraction of the mesh .... 48
3.4 The two Parallel Efficiency, or ‘scaling factor’ equations for MG-CFD and CPX ................................................................. 54

4.1 Rolls-Royce (RR) Coupler ~10M Cell Tests (Test Case 1) ............ 65
4.2 CPX ~10M Cell Tests (Test Case 1) ........................................ 66

5.1 The list of mesh sizes used in the mini-simulation. ....................... 90
5.2 A comparison between the pressure solver test cases and the equivalent SIMPIC test cases (Base-STC) which replicate the performance behavior. 91
5.3 The parallel efficiency, or ‘scaling factor’ equations for the Base SIMPIC Test Case (Base-STC) and Optimized SIMPIC Test Case (Optimized-STC) 104
5.4 The mesh size and MPI ranks of each component in the small 3-stage mini-app simulation. ....................................................... 109
5.5 The mesh size and rank allocation of each component in the full 16-stage mini-app simulation .................................................... 112
Acknowledgments

Throughout my research journey over the last few years, I have been guided, helped and encouraged by many people, of which I express endless gratitude. I have had a brilliant time working with wonderful people, and will look back on the last 4 years with fond memories.

I would like to thank Prof. Stephen Jarvis for offering me this opportunity and helping me work out the focus of the work over the first 9 months of the project. Even after his move to greater things at the University of Birmingham, his support has been extremely valuable. I would like to express significant appreciation to my supervisor for the majority of my studies, Prof. Gihan Mudalige. He has given near limitless support, guidance, and patience for which I will be eternally thankful. Additionally, I would like to thank Dr. Matthew Leeke and Dr. Richard Kirk for both support and many hours of interesting discussions.

I am grateful for the support of my friends and colleagues, past and present, in the Warwick High Performance Systems Group, including Dr. Arun Prabhakar, Suneth Ekanayake, Alex Cooper, Andrew Lamzed-Sort, Sam Curtis, Joe Kaushal, Dr. Dean Chester, Dr. Dominic Brown and Dr. Andrew Owenson. I would also like to thank Kabir Choudry, Zhihao Dai, and Dr. Kamalavasan Kamalakkannan, with whom I taught systems programming labs in the early and later phases of my research. The many jokes and conversations I had with them over the years made the long breaks between student questions fly by.

I would like to show my appreciation to my friends from my undergraduate days, in particular Sam and Natasha, who are some of the kindest and smartest people I know. I would also like to thank my family, in particular my Mum, Dad, and Granddad Colin, as well as my partner’s parents, Rita and Kenny, who have all been incredibly supportive. Finally, my greatest appreciation goes to my partner Sophie, without whom I wouldn’t be where I am today. She is the love of my life and I dedicate this thesis to her.
Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree.

Several components of this work have been published by the author:

Parts of Chapter 2 and Chapters 3 and 4:


Parts of Chapter 2 and Chapter 5:


Other work that has been completed but not included in this thesis:


Mini-coupler code is available on GitHub:

- github.com/warwick-hpsc/CPX

Name: Archie Powell

Date: 25-09-2023
Abstract

As the complexity of multi-physics simulations increases, there is a need for efficient flow of information between components. Discrete ‘coupler’ codes can abstract away this process, improving solver interoperability. One such multi-physics problem is modeling a gas turbine aero engine, where instances of rotor/stator CFD and combustion simulations are coupled. Allocating resources correctly and efficiently during production simulations is a significant challenge due to the large HPC resources required and the varying scalability of specific components, a result of differences between solver physics. In addition, configuring couplers and allocating resources correctly can be challenging for such problems due to the sliding interfaces between codes in the compressor and turbine. In this work, we present CPX, an open source mini-coupler designed to model the performance behavior of a production coupler framework at Rolls-Royce plc., used for coupling rotor/stator and combustion simulations. We demonstrate high qualitative and quantitative predictive accuracy with a less than 17% mean error. A performance model is developed to predict the ‘optimum’ configuration of resources, and is tested to show the high accuracy of these predictions. Using these tools, we develop a coupled mini-app simulation, analyzing the performance bottlenecks and examine a selection of optimizations, allowing us to estimate the workload’s performance. This coupled simulation is supported by the performance model which is then used to load balance and predict the speedup of a compressor-combustor-turbine simulation of 1.2Bn cells, a production representative problem size. The model is validated on 40K-cores of an HPE-Cray EX system, predicting the run-time of the mini-app work-flow with over 75% accuracy. The developed coupled mini-apps and performance model model combination demonstrates how rapid design space and runtime setup exploration studies can be carried out to obtain the best performance from full-scale CFD-CFD and Combustion-CFD coupled simulations.
Abbreviations

**ADT** Alternating Digital Tree

**AMG** Algebraic Multigrid

**AoS** Array-of-Structs

**Base-STC** Base SIMPIC Test Case

**CESM2** Community Earth System Model 2

**CFD** Computational Fluid Dynamics

**CPU** Central Processing Unit

**CPX** Coupler Extensions

**CU** Coupler Unit

**DLR** Deutsches Zentrum für Luft-und Raumfahrt

**DS** Density Solver

**DSL** Domain Specific Language

**ECP** Exascale Computing Project

**EPCC** Edinburgh Parallel Computing Centre

**FEM** Finite Element Model

**FSI** Fluid-Structure Interaction

**GPU** Graphics Processing Unit

**HPC** High Performance Computing

**HPCp** High-Pressure Compressor

**HPT** High-Pressure Turbine

**IP** Intellectual Property
IPC  Intermediate-Pressure Compressor
LES  Large-Eddy Simulation
LoC  Lines of Code
MG   Multi-grid
MPI  Message Passing Interface
MUM  Multi-Unit Mode
Optimized-STC  Optimized SIMPIC Test Case
PCle  Peripheral Component Interconnect Express
PDF  Probability Distribution Function
PIC  Particle-In-Cell
PS   Pressure Solver
RANS  Reynolds-averaged Navier–Stokes
RR   Rolls-Royce
SIMD  Single Instruction Multiple Data
SPA  Sparse Accumulator
SpGEMM  Sparse General Matrix-Matrix Multiplication
SPMD  Single Program Multiple Data
SpMV  Sparse Matrix–Vector Multiplication
TDP  Thermal Design Power
URANS  Unsteady Reynolds-averaged Navier-Stokes
WDM  Whole Device Model
Chapter 1

Introduction

Concurrent execution of multiple physical models as a single simulation has emerged as an important approach for modeling large-scale multi-physics phenomena. This strategy, as opposed to carrying out a single monolithic simulation, allows for decomposing complex systems into a series of smaller, interconnected components, which can utilize the optimal method for modeling the physics of each sub-domain. Not only does this approach simplify code development and maintenance, but also allows domain scientists to select the optimal numerical methods and architectural optimizations for modeling different physical environments.

The challenge with such a modular approach is the efficient flow of information between the multiple models through common interfaces that does not lead to (1) numerical errors that a non-coupled or monolithic simulation would not have caused and (2) performance bottlenecks degrading the time-to-solution or throughput. A coupler acts as a discrete piece of code dedicated to implementing this flow of information and its design crucially determines the performance of the full simulation. For the domain of Computational Fluid Dynamics (CFD), coupled simulations provide flexibility to combine specialized flow solvers and/or different turbulence models. Such coupled simulations are currently common practice in industry. Examples include the coupling of incompressible flow solvers for modeling flow within a combustion chamber with the compressible flow in the outlet vane [Kannan and Page, 2014] or using a hybrid Reynolds-averaged Navier–Stokes (RANS)/Large-Eddy Simulation (LES) modeling approach where the problem domain is
decomposed and different turbulence models are applied depending on flow conditions of each zone [Fröhlich and Von Terzi, 2008; Schlüter et al., 2005b].

The most complex coupling scenarios consist of close interaction of CFD models with other numerical simulations, including Finite Element Model (FEM) and combustion, for real-world systems. One such example [Arroyo et al., 2021] is modeling a gas turbine engine, consisting of multiple high-pressure compressor and/or turbine stages interacting with the combustion chamber. In this setup, cold air entering the engine is compressed before it is delivered to the combustion chamber, where it is sprayed with fuel and ignited. The exhaust air from the resulting combustion provides the thrust that drives the turbines which, in turn, spins the compressor and fan. Currently, the components are designed and manufactured separately, which can lead to reduced engine performance due to optimizations made within individual elements being negated as a result of integration with other components. This is because without coupled simulations, there is information loss as unsteady interaction between components cannot be modeled [Arroyo et al., 2021].

Within coupled simulations, solvers interact with each other by transferring some “interface” or “boundary” data to its neighboring simulation via the coupler(s). The couplers map the values, or fields from one simulation to the other, interpolating data / translating the information as required [Larson et al., 2005]. Naturally, adequate resources should also be allocated to the coupler components for efficient performance, aiming for a smooth overall simulation without performance bottlenecks due to the information exchange. This comes in addition to the more common challenge of balancing resources for each simulation, where maximum concurrency needs to be achieved without idling components awaiting data. As such, a key factor in developing and carrying out performant coupled simulations is the design and execution space exploration required to obtain the optimum load and resource balance between the multiple simulations and couplers. However, given the scale of production executions including long execution times and the need to use large High Performance Computing (HPC) systems, a direct, brute force tuning of parameters, load balancing and run-time configuration is prohibitively expensive and laborious.

The underlying aim of this work is to analyze and address several significant challenges of running efficient coupled multi-physics simulations on modern HPC systems within the realm of gas-turbine aero engines. We build a mini-coupler framework and accompanying empirical performance models to predict the best allocation of resources and determine the
bottlenecks within a large scale CFD-Combustion simulation on modern HPC hardware.

1.1 Motivation and Problem Statement

Coupled simulations have become furthermore important with the industry’s ambitions to reach virtual certification of full aero-engine designs [European Commission, 2011] [ASiMoV, 2018]. Ultra-high fidelity simulations are required, pushing current model sizes of 10-100 million elements to tens of billions of elements.

![Image of RR Trent XWB Engine](Rolls-Royce, Accessed May 2022) (Reproduced with Permission)

The challenges of running such coupled simulations are significant. Consider the compressor stage of a Rolls-Royce gas turbine engine, shown in Figure 1.1, which is made up of many blade-rows attached to a shaft which spins at high speed to compress air entering the front of the engine. Its function is to increase the air pressure, forcing the high pressure air into the combustion chamber. Although the simulation of a compressor can be ran using a monolithic approach, it is significantly (up to $2.8 \times$) more efficient to use a coupled method [Mudalige et al., 2022]. Coupled simulation of the full compressor therefore consists of a number of connected RANS or hybrid RANS models representing simulation of rotor and stator blades. Each of the rotors and stators are typically simulated using instances of an unstructured grid, density-based, explicit CFD solver [Lapworth, 2004].

The Density Solver (DS) instances are linked together by custom Coupler Unit (CU)
instances as illustrated Figure 1.2. The custom coupling configuration is then set up such that the DSs operate on distinct meshes that cover adjacent or overlapping zones of the physical space [Amirante et al., 2021b]. On a parallel HPC system, distributed memory parallelism is used (specifically Message Passing Interface (MPI)), where each DS is assigned a number of MPI processes. A CU is also allocated a number of processes to carry out the inter-linking between the DSs. In order to simplify the interfaces, improve parallelism and in-turn performance, a single CU will only manage one geometric interface, namely the interface (or part of the interface) shared by two DSs. This means that the interface shared by the two DSs can be allocated multiple CUs, each handling a segment of the interface. Consequently this leads to a complex configuration of DSs-to-CUs connectivity and MPI processes to DSs and CUs, that for a given overall simulation, require careful orchestration of distributed memory resources to obtain optimal performance. As each rotor moves through every time step, sliding plane interfaces must be used, where the mapping between the stator and rotor interfaces must also be recomputed for each time step. This process, performed by the CUs, can significantly degrade the performance of the coupled system if ranks are not allocated correctly.

These problems become even greater when modeling the complete engine core, consisting of compressor, combustion chamber, and turbine components, as seen in Figure 1.1. Here, the compressor and turbine are typically modeled the same density solver as previously mentioned, with the combustion chamber employing a LES turbulence model with Lagrangian fuel spray [Anand et al., 2013], and a pressure-based, implicit CFD solver. As before, the rotor/stator interaction in the density solver uses the sliding planes method, but a steady-state approach is used to model interaction between the density and pressure solvers [Amirante et al., 2021b]. This brings new challenges such as different solver time-step lengths, execution times, strong and weak scaling characteristics, and frequencies of recomputing coupler unit interfaces. Given the different coupling interfaces and interactions between components, the scale of production executions, including long execution times (in the order of days or weeks) and the need to use large HPC systems, a direct, brute force tuning of run-time parameters and load balancing is prohibitively expensive and unachievable under production settings.

In this work, we attempt to develop techniques to solve the above design space and configuration exploration problem. We first develop a stripped-down, “proxy” coupler
Figure 1.2: An annotated model of the DLR Rig250 compressor test case (top) and setup of a Rotor/Stator test case with Density Solver instances (DSs) and Coupler Units (CUs) (bottom) [Marciniak et al., 2014] (Reproduced with Permission)

framework called Coupler Extensions (CPX), producing a tractable, yet representative application to aid the investigation. The idea follows on from the widely used technique in HPC where simplified versions of large applications, called mini-apps, are used to explore co-design, performance and optimum configurations of applications on HPC systems [Pen-nycook et al., 2013; Reguly et al., 2015]. To build a proxy coupled CFD simulation, modeling an engine compressor, two mini-apps are required - a coupler mini-app and CFD mini-app. Rolls-Royce already uses an open-source mini-app called MG-CFD [Owenson et al., 2020] to assist the development of their density solver CFD application. As such, our focus in this work is the equivalent coupler mini-app. By combining instances of MG-CFD together using CPX, and developing an empirical performance model, we investigate, model and predict the optimum DS-to-CU connectivity/configuration, aiming to understand how to best allocate resources for full-scale production simulations in the compressor of a gas turbine.
From here, we create a simplified version of a full scale Compressor-Combustor-Turbine simulation using scaled-down, but representative applications. Building on the coupled mini-app compressor with MG-CFD [Owenson et al., 2020] and CPX [Powell et al., 2021] as the proxy for the density-solver and coupler respectively, a new pressure solver proxy modeled by the SIMPLIC [SIMPLIC, Accessed May 2022] Particle-In-Cell (PIC) code is added to create the compressor-combustor-turbine triple-components as a coupled mini-app simulation. The empirical performance model is improved and extended to reason-about and explore the optimal resource allocation for the coupled mini-apps when executing a large mini-app engine simulation. We then use the analysis and predictions from the mini-app simulation and performance model to predict the optimum allocation of resources for the full-scale simulation of the 1.5B High-Pressure Compressor (HPCp)+Combustor+High-Pressure Turbine (HPT) test case using production codes.

The primary objective of the work is to predict the best allocation of resources for running coupled CFD/combustion using mini-apps, as there can be a significant bottleneck on the parallel efficiency of the simulation if resources are not adequately distributed. Using the full production codes will waste HPC resources and due to the scale required to run, the problem quickly becomes intractable when reaching current production problem sizes that are at 1B to 5B cells. The challenge of configuration exploration becomes even greater as the amount of compute resources increases. In addition, we aim to determine the bottlenecks for the workload in the codes themselves and examine optimizations, speculating what the optimized run-time would be. Similarly, without mini-apps, it becomes difficult and time consuming to achieve due to the size and complexities of the production codes. Addressing these bottlenecks represents an important step as part of an ongoing push towards virtual certification.

1.2 Thesis Contributions

More specifically, we make the following contributions:

- We present CPX, a representative mini-app designed to match the performance behavior of Rolls-Royce (RR)’s production coupler framework. CPX is combined with the MG-CFD mini-app to create proxy configurations to capture the operation
of the production compressor simulations. The run-time and scaling behavior of MG-CFD with CPX is shown to have the same quantitative and qualitative behavior as Rolls-Royce’s production applications.

- An analytical comparison is used to demonstrate that for a certain allocation of coupling resources, CPX and RR Coupler will have similar scalability behavior. CPX + MG-CFD is then tested using a variety of configurations, and compared to RR Coupler + RR Density Solver, showing that proxy apps can predict the run-time with a less than 17% mean error.

- Using the insights from CPX and MG-CFD, an empirical performance model is created to predict the best resource allocation for a given coupled compressor simulation of the production applications. The model is used to predict optimum configuration settings on a cluster, and the prediction is then tested against other configurations on the 740,000 core ARCHER2 supercomputer [EPCC, Accessed May 2022] to show that the predicted configuration is optimal. Finally, the model is used to predict the optimum configuration for a 4.6Bn cell test case using 100,000 cores, as well as the same run on a 400 Graphics Processing Unit (GPU) system.

- The Particle-In-Cell mini-app, SIMPIC [SIMPIC, Accessed May 2022], is used as a black-box performance proxy to model the behavior of the combustor pressure solver typically found in aero-engine simulations. This Base SIMPIC Test Case (Base-STC) is tested using a variety of configurations, predicting the pressure-solver run-time with a max error of 22%.

- A selection of particle and multi-grid solver optimizations are explored, and their contribution to performance are estimated theoretically, if applied to the production pressure-solve. An Optimized SIMPIC Test Case (Optimized-STC), is created from SIMPIC which synthetically matches this theoretical performance.

- SIMPIC is then coupled with multiple instances of the MG-CFD mini-app together with the CPX mini-coupler to create a coupled mini-app compressor-combustion-turbine simulation. The empirical performance model is extended to include the pressure solver and its coupling. This allows us to load balance and predict the
speedup of the complete mini-simulation. Both the Base-STC and Optimized-STC versions are modeled.

- The model predicts the run-time of each individual component of the coupled mini-app simulation, each with less than 25% error. Furthermore, the large coupled mini-app simulation of the HPCp-Combustor-HPT test case is executed on up to 40,000 cores on the ARCHER2 (HPE-Cray EX) supercomputer comparing run-times with predictions from the performance model. Predictions are over 75% accurate for the both Base-STC and Optimized-STC coupled versions.

### 1.3 Thesis Limitations

Similar to other works investigating application performance [Mudalige et al., 2008] [Bunt et al., 2016], this research employs a performance model to estimate the run-time of applications based on a number of parameters. Our method does not use an analytical approach, instead favoring a method using benchmarking and statistical analysis. While the technique has limitations, the principal being that error can be higher than using an analytical approach, it is often used in load-balancing in high performance computing [Berman et al., 1996] and still delivers accurate predictions that can be used with the other techniques presented in this work. We believe as the complexity of coupled simulations increase and the scope of optimization grows, there is opportunity for performance predictions that do not rely on the creation of analytical models, which traditionally take significant time to develop [Escobar and Boppana, 2016].

When measuring application performance, we only focus on the run-time of the application and ignore the setup cost. While at small scale this is cost is non insignificant, the full-scale coupled simulations that are the focus of this work run for many hours on tens of thousands of cores, with the setup cost making up only a very small part of the final run-time. Additionally, aside from a brief comparison of GPUs and Central Processing Unit (CPU)s, we do not examine power consumption in our predictions. In running a high performance application, it is usually beneficial to allocate as many resources as possible as while parallel efficiency remains high [Mudalige et al., 2013]. In this work, resources are only allocated to applications while parallel efficiency is above 50%, and the
techniques shown in this research can be readily extended to include power consumption if required.

1.4 Thesis Overview

This chapter details a brief background, problem statement, and motivation for the work behind the thesis. The remainder of the thesis is structured as following:

- Chapter 2 highlights use of code coupling in multi-physics simulations and breaks down the different approaches that can be taken. The methods of Rolls-Royce’s internal coupling framework, and their relation to gas-turbine engine simulations are detailed, along with its limitations. The role of mini-apps is laid out, both in general and in the representative industry-driven problems we seek to address in this thesis. A brief description of performance modeling, and where the efforts in this work lie in the context of wider research is presented.

- Chapter 3 details the development process and design of the CPX mini-app, including comparisons with the industry coupler code. The development an accompanying empirical performance model is also presented, along with an iterative algorithm in which it is implemented. Finally, the large-scale production-grade problem we intend to replicate is examined and a mini-app equivalent test case is specified.

- Chapter 4 explores the capabilities of the mini-coupler and performance model, comparing the predicted performance and resource allocation with the industry simulation across a range of test cases. Using these tools, a limitation of the industry coupler is identified, and the tools are updated to account for the required changes. Finally, the predictive capabilities of the empirical model are validated for a large 4.6Bn cell test case.

- Chapter 5 describes the extensions required for running a full-engine mini-simulation in CPX, including integrating SIMPIC, modifying the input mechanism, and the necessary adjustments to mini-apps to improve stability. The setup and challenges of the full-engine simulation is examined, and the individual components of the simulation are compared to determine which will be the bottleneck. This component,
the pressure solver, is profiled and a number of optimizations in wider research are investigated, with the results from these optimizations being extrapolated to estimate their effect on an optimized pressure solver. The extensions in the empirical performance model are also explored, including an improved iterative algorithm that can allocate resources to different solver and coupler types, as well as different mesh and boundary input sizes. From here, the use of the mini-coupler and extended performance model with different solver and coupler types is demonstrated, in both a small 3-stage test case and a large 16-stage full-engine mini-simulation of 1.5Bn cells. The limitations of the original pressure solver are explored, and the predictive capabilities of the extended empirical model are validated against the mini-simulation.

• Finally, Chapter 6 summarizes the thesis, examining the motivations and contributions of the research, and exploring avenues of future work.
Chapter 2

Background and Related Work

When modeling aero-engine design using numerical simulations, traditionally their scope is limited to individual segments/components of the engine such as the compressor or combustion chamber. Interactions between different sections are handled using boundary conditions, often taken from the time-averaged solution of the adjacent components. However, information is often lost as a result of this process, as bidirectional transfer of some information, particularly unsteady interactions such as turbulence intensity, are not possible. Therefore, it is desirable to run multiple components together in a single simulation such that interactions can be more accurately captured [Arroyo et al., 2021]. For the domain scientists, these coupled simulations provide the flexibility to select, for instance, the best numerical method and problem scale for each component, or even parallelization/target architecture to execute the sub-components. It also significantly simplifies code maintenance and extension essentially implementing a horizontal separation of concerns approach, akin to the “vertical” separation of concerns achieved by Domain Specific Languages (DSLs) [Cleenewerck and Kurtev, 2007] in HPC, where expertise in developing different simulation models can be leveraged to gain the best results.

2.1 Code coupling

The process of coupling codes usually involves four core stages. Firstly, interfaces which sit between simulations are defined, and are used as the domain in which data is transferred.
These interfaces are made of slices of the original domains, and by ensuring that the values across the interfaces match, the system remains stable. This can be done as a pre-processing step or at run-time. Next, a mapping is created which links the cells in the interfaces, so when data is transferred (e.g. for every iteration of a solver) the cells of the interface mesh know where to send and receive data. These first two stages are usually only performed once.

Once the solvers are running, they transfer data between each other during each iteration. When data is received, it is interpolated to ensure the values remain consistent across the interfaces. This interpolation is necessary as the interfaces may not align, thus the values for a particular interface cell will be a combination of multiple cells from the other interface, as well as its prior value. This can be seen in Figure 2.1. The ‘communicate and interpolate’ process is repeated for every iteration of the solver, until the simulation concludes.

Traditional code coupling is a well-studied field, and a variety of frameworks exist to aid the implementation of these stages. These frameworks vary in scope and function;
Figure 2.2: A comparison highlighting the issues of peer-to-peer coupling when the coupling routines are computationally heavy

frameworks such as MUI [Skillen et al., 2020] and preCISE [Bungartz et al., 2016] act purely as an interface where data can be sent and retrieved, whereas others, such as MCT [Larson et al., 2005], are more involved, with dedicated classes for data fields and methods for interpolation and other transformations.

A core requirement for any coupling setup is to ensure the parallel performance of each solver, and by extension the simulation, is not compromised as a result of the coupling implementation. As a result, both the solvers and coupling framework must distribute work to a group of processes. In the examples mentioned above, the interface data is transferred directly between solver instances, with the coupling stages being performed by the resources allocated to each solver. This strategy is popular as it offers a good balance between parallel performance and simplicity. However, the performance can degrade if a significant amount of computation is required during the coupling stages, or if the interacting solvers have different scaling or convergence characteristics.

Consider two solvers, A and B, coupled together in a simulation. The solvers have the same strong and weak scaling, and take the same time per iteration, but Solver A takes 10 iterations to converge, whereas Solver B takes 40 iterations to converge. With any bidirectional coupled simulation, a solver must wait for another interacting solver to
Using a distributed client server-coupling model, the run-time can be reduced if the coupling work is computationally heavy.
and finish each time-step at the same time. This reduces the time the solvers are spent being idle, and can be further reduced by overlapping some of the coupling work with the solvers [Ganine et al., 2015]. Regardless of the chosen framework, the significant challenges of resource allocation in coupled codes is well documented [Amirante et al., 2021a]. Depending on the coupling approach, the exchange of information between solvers may contribute significantly to the run-time of the simulation.

2.2 Coupling in multi-physics problems

As with coupled simulations of gas-turbine aero engines, load balancing and understanding performance bottlenecks within coupled multi-physics problems remains an ongoing challenge in HPC [Keyes et al., 2013; Totounferoush et al., 2019]. We will examine a selection of such problems to understand the applicability of methods to load balance and gain insight via coupled mini-apps.

2.2.1 Fluid-Structure Interaction

Fluid-Structure Interaction (FSI) problems focus on the flow of fluid, typically calculating its pressure and velocity using Navier-Stokes and its interaction with a geometric description of a structure where displacement is modeled [Keyes et al., 2013]. Examples include analyzing the airflow around a parachute and the effects on the material [Boustani et al., 2019], and the structural rigidity of aircraft [Farhat, 2004]. Traditionally, coupled problems in this domain would be ran loosely-coupled in lock-step, where the fluid would update, exert a force on the structure, which then updates the flow field [Keyes et al., 2013]. In recent work, significant work has been undertaken to run these schemes in parallel; however, due to the overhead of coupling and the significant difference in computational intensity between fluid and structural solvers, load balancing and optimizing these simulations remains an ongoing challenge [Mehl et al., 2016].

2.2.2 Fission Reactor Fuel

In a nuclear fission reactor, fuel pellets, made up of missile material and surrounded by protective cladding, are used to transfer energy into water which can be used to drive a
steam turbine [Brain and Lamb, 2000]. One critical area of research is the interaction between the fuel pellets and the cladding, in order to improve the structure and longevity of the fuel system [Hansen et al., 2009]. Simulating these interactions is challenging, involving many coupled equations such as nonlinear heat conduction, contact, radiation damage, and structural deformation [Gaston et al., 2013]. The latest simulations with advanced solver libraries reached 77% parallel efficiency using 32,768 2nd-generation Intel Xeon Phi cores on the Theta supercomputer at Argonne National Laboratory [Kong et al., 2018; Permann et al., 2020], demonstrating that massively parallel reactor simulations are within reach. Despite this, there is important work to be done to keep up with the rapidly advancing hardware HPC landscape.

2.2.3 Magnetic Confinement Fusion

Power generated from nuclear fusion has the potential to transform our energy space and accommodate our growing power generation requirements without the increase use of fossil fuels [Toschi, 1997]. These reactors work with the use of a tokamak, a device that generates a magnetic field strong enough to hold plasma in the shape of a torus, required to reach temperatures to enable fusion of atomic nuclei. Simulating nuclear fusion is complex, in part due to the difficulty of modeling the interaction between the plasma in the tokamak and the wall of the reactor [Linke et al., 2019]. Current state-of-the-art research has been focused on coupling physics codes simulating the core and edge regions of the tokamak [Merlo et al., 2021]. These represent the foundation of a reactor, with the aim to achieve a Whole Device Model (WDM), coupling equations describing many different physical phenomena. As these are incorporated, the complexities of the simulation and coupling will grow, and the number of parameters will increase [Linke et al., 2019]. Thus, there is clear scope for performance modeling tools, mini-apps and methods to gain insight without using large scale applications.

2.2.4 Climate Modeling

Modeling the earth’s climate is particularly challenging due to both the time-scale and area of the earth’s surface, as well as the number of coupled components within a system [Keyes et al., 2013]. The most advanced climate model, the Community Earth System
Model 2 (CESM2), couples together simulations of sea ice, land ice, land, ocean, surface waves, river run-off and atmosphere to provide an overview of the earth’s climate [Danabasoglu et al., 2020]. The complexity of the coupled system highlights the advantages that a coupled simulation of mini-app components could bring. One example was the model forming unrealistic amounts of sea-ice around the Labrador Sea area in Greenland during development, something that could not be linked to any particular component despite extensive analysis of simulations [Danabasoglu et al., 2020]. A coupled mini-app approach could aid in understanding where the source for the error originated. Additionally, load balancing each of the components has been well documented as a highly demanding task, since each of the components must be allocated their own compute resources [CESM, Accessed July 2023]. Predictive resource allocation methods have already been demonstrated in earlier versions of the model [Alexeev et al., 2014], and are becoming ever more important as the complexity of the model increases.

2.3 Coupling in a gas-turbine aero engine

As illustrated in Figure 1.1, the core of gas-turbine aero engine can be split into three distinct components: the compressor, the combustion chamber, and the turbine. We will first examine the compressor, which from a coupling perspective is very similar to the turbine, then examine the limitations with the RR coupler, before including the interactions with the combustion chamber and examining the simulation as a whole.

2.3.1 Coupling in a gas-turbine compressor

The compressor stage of a gas-turbine engine is made up of a series of blade rows, called rotor blades and stator blades, as seen in Figure 1.2. The rotor blades are attached to a central shaft and rotate at a certain angular velocity, whereas the stator blades are fixed to the casing of the engine and are stationary. The flow fields of the rotors and stators are simulated using instances of the Rolls-Royce density solver application [Lapworth, 2004], an unstructured-mesh application which uses a 5-step Runge-Kutta method for time-marching, accelerated by multi-grid and block-Jacobi preconditioning. Each blade row is represented by a separate mesh, with these meshes being coupled together to
produce a complete simulation using the process described in Section 2.1. The aim of the coupling process, in particular the interpolation stage, is to provide a smooth transition between each of the blade rows, achieved by averaging the values of the flow field on the boundaries between the two domains. By doing so, any unsteadiness between boundaries is removed [Manual, 2009].

Figure 2.4: With mixing plane coupling, the rotor/stator interaction is treated as a steady-state problem and the rotor does not move relative to the stator.

Figure 2.5: With sliding plane coupling, the rotor is moving relative to the stator requiring the mapping between the two boundaries to be recomputed every time-step.
In the simplest form of code coupling, known as steady-state, the coupled domains are all stationary. Interaction between the two domains is simple as it will always be known which of the cells in the domains will interact with one another. While each rotor blade row in a compressor moves relative to the stator row, this isn’t necessarily an issue, as it is possible to treat each interface as a steady-state problem and spatially average the flow field data, known as a mixing-plane [Manual, 2009]. Figure 2.4 shows this approach. Two domains are presented, a static stator grid, with black nodes, and a rotating rotor grid, with white nodes. Each domain has a dashed area which overlaps the opposite domain, which we refer to as the interface or boundary. This interface is used to average values (e.g. momentum) between the two domains. At each time-step, the value of each cell in the interface is updated by averaging the current cell value, the value of the cell at the new time-step, and the values of neighboring cells. The mixing-plane interface is fixed, so when the simulation moves forward a time-step, its position doesn’t change. If the two domains weren’t moving, then the values would be time-accurate, because their position wouldn’t change with time. However, since the rotor is moving, the solution will be time-averaged. The aim of the simulation is to model the unsteady interaction between the blade rows, requiring a time-accurate solution, thus a mixing-plane cannot be used. Instead, a sliding-plane approach must be used.

Figure 2.5 presents a sliding plane approach. Unlike a mixing-plane, where the interface is static, in a sliding plane the interface moves with the rotation of the rotor blade. This allows the interface to capture a more accurate representation of the interaction between the stator and rotor blade. When the values between the two domains are interpolated, the interface must calculate which cells overlap, as overlapping cells are used to calculate the updated value at each time-step. The process is intensive, as each cell in an interface needs to compare itself with every other cell in the other interface, and must be repeated for all interfaces in a coupled simulation. In a mixing-plane, the interface is fixed in place, so the computation only needs to be computed once at the start of the simulation. Thus, while a full scale simulation may be thousands of time-steps long, the overall cost is insignificant. However, with a sliding-plane, the mapping which links the cells in one interface to the other must be recomputed every time the interface moves. In
the context of a series of coupled rotor/stator chambers, such as those shown in Figure 1.2, this recomputing needs to be performed every time the rotor blade(s) rotate. The recalculation, consisting of a search routine among other operations, incurs significant computational overheads [Powell et al., 2021], which we will explore in Section 2.3.3.

2.3.2 Coupling in a full gas-turbine engine

As the combustion chamber and the stators of the compressor and turbine are stationary, the interaction is modeled as steady-state. This may imply that the interaction with the combustion chamber is trivial, however this may not be the case. As previously mentioned, the Rolls-Royce combustion application employs a LES turbulence model with Lagrangian fuel spray [Anand et al., 2013], and a pressure-based, implicit CFD solver. This differs from the Unsteady Reynolds-averaged Navier-Stokes (URANS) method that the density solver uses to model the compressor and turbine. While this is faster than using a LES throughout, URANS-LES interaction does result in approximation of some quantities, which can result in significant instability within the pressure solver if the fidelity of the coupling is not sufficient. This requires the interface to be an order of magnitude larger than in the sliding planes, and in some instances requiring overlapping the boundaries of the mesh, similar to using ‘overset’ methods for modeling rotor/stator interaction [Ganine et al., 2015]. While the interpolation stages of coupling are less intense than the search, the large interface size may still result in load-balancing issues if resources are not properly allocated.

The URANS-LES approach mentioned is the current standard for modeling these components together [Arroyo et al., 2021]. While this method has been applied to a variety of coupled gas turbine models [Jacobi et al., 2017] [Salvadori et al., 2012], accurately coupling together all three components (compressor, combustor, turbine) remains a key challenge. The first published example, by Stanford University’s Center for Turbulence Research, use a LES approach for the Combustor and URANS for the compressor and turbine. However, due to the difficulty in coupling the codes, the work reports inaccuracies with the generated results [Schlütter et al., 2005a]. The first mathematically accurate example was performed by NUMECA, simulating compressor-combustor-turbine interaction in a KJ66 gas turbine engine [Romagnosi et al., 2019]. However, this was achieved using a RANS
approach throughout the simulation, which is a much lower accuracy method compared to a URANS or LES model that is typically needed for production aero-engine design. Furthermore, the engine simulated was two orders of magnitude smaller in length than a typical aero-engine found on commercial aircraft [Saab Gripen, 2013].

More recently, a team at CERFACS successfully ran a coupled three component simulation, modeling the fan, compressor, and combustion chamber of a DGEN-380 turbine engine [Arroyo et al., 2021]. This simulation used a LES approach throughout, which ensures very high accuracy. However, the solution requires an initial simulation of a group of smaller components as stand-alone simulations; the integrated/coupled simulation cannot finish without this. Nevertheless, their work provides a good point of reference as the mesh size used is very similar to the test case we will examine later in this thesis (2.1Bn cells), albeit with fewer components.

The numerical stability of the system is not the only challenge of coupling a gas-turbine engine. An important distinction between the Rolls-Royce density solver and pressure solver, which are used to model the fan blades and combustion chamber respectively, is that the density solver currently uses explicit methods [Lapworth, 2004] whereas the pressure solver uses implicit methods [Anand et al., 2013]. These methods are used to obtain an approximate solution to the differential equations which model the physical process, which in a gas-turbine engine is primarily the airflow. This process is performed every time-step in the simulation. When the approximation is required for the next time-step, explicit methods generate a solution to these equations based on the results from the current time-step, whereas implicit methods generate a solution using both the current time-step approximation and the next.

Explicit solvers are much easier to implement than implicit solvers, however they are not always suitable for a given problem as they have a narrower window of stability than implicit methods. Implicit solvers also converge much faster to the correct solution, which can allow for greater performance. An implicit solver is needed for the pressure solver as using explicit methods in a LES results in very poor performance unless time-step size is impractically small [Shyy et al., 1992]. The result of these differences, a consequence of the solver models (RANS vs LES), is that the size of a time-step in the density solver is twice as long as in the pressure solver. As the entities in a bidirectional coupled system must be advancing at the same rate, this requires the number of pressure solver time-steps
to be doubled, and by extension also the run-time of the pressure solver instance.

This has implications when considering how to best allocate resources and identify bottlenecks in a coupled gas-turbine simulation. Firstly, each of the solver instances have varied mesh and interface sizes, and interaction between different types of solver may require different coupling types, as detailed in Section 2.3.1. The process is then further complicated by the mismatched time-step lengths which require the number of time-steps to be scaled up or down to match the pace of the system. There are also ramifications when considering optimizations; some changes, such as switching the density solver method from explicit to implicit, will result in a different time-step length which could further boost or weaken performance. Importantly, optimizing a particular solver will only reduce the run-time of a coupled simulation if a solver instance of that type is the slowest/least efficient in the system. Even with just a single solver type, addressing these problems already requires significant HPC resources [Powell et al., 2021], and as the number of interacting solvers and mesh sizes increase, it becomes intractable to do without tools which can help capture and evaluate the performance of components, predict run-time, and highlight where optimization efforts should be placed.

2.3.3 Design of the RR Coupler

As previously stated, the RR Coupler, also known as JM76, is a decentralized, client-server [Amirante et al., 2021b] coupling framework. Coupling is performed via dedicated compute resources called Coupler Units (CUs), which the Pressure Solver (PS) or DS (Density Solver) instances send and receive boundary information to and from.
Figure 2.6: The three core stages of a simulation using the RR Coupler [Amirante et al., 2021b]

Figure 2.6 shows the three core stages of a simulation built using the RR Coupler. The RR Coupler binary is the starting point of a simulation, which initializes libraries and reads the user specified parameters which define the layout of the simulation. The framework then begins the setup stage, creating instances of each solver by splitting the main MPI communicator into sub-communicators and passing this into the initialization stages of each solver. This requires each solver to be built as a library and linked to the RR Coupler at compile time. Data structures are created telling each solver which coupler units it must communicate with every time-step, and tell the coupler units which solvers they are communicating with. MPI ranks can either be allocated to a single coupler unit, or many coupler units can be created, each with a single MPI rank. We will explore the differences of these approaches in Section 2.3.3. Finally, there is the execution stage, where the solver routines run, communicate with the coupler units each time-step, and the coupler units perform their coupling routines.
Limitations of the RR Coupler

In a brute force search, the number of memory accesses per interface is equal to the square of the size.

In the RR Coupler, the first stage of coupling (defining the interfaces) is performed before the code is run. The other three stages (communication, search, and interpolation) are performed by the CUs. Each density/pressure solver session runs for a set number of time-steps, in which the rotor rotates every time-step, and between each time step the solver runs for a certain number of iterations. At every time-step in the compressor or turbine, the CU(s) run a search routine to recompute the mapping between the interface cells. During solver iterations, the data is transferred from the solver sessions to the CU(s), where the CU(s) interpolate the data such that the values are consistent across interfaces.

To understand the limitations of the RR Coupler, we will examine how the search routine, which must be performed every time-step in compressor and turbine simulations, is performed within the code. As previously mentioned, in a sliding plane the mapping which links the cells in one interface to the other must be recomputed every time the interface moves, i.e. every time the rotor blade(s) rotate. The process is intensive, as each cell in an interface needs to compare itself with every other cell in the other interface, and must be repeated for all interfaces in a coupled simulation, both moving and static, illustrated in Figure 2.7. As a result of this ‘brute force search’, if the average interface size
doubles, the search run-time increases by a factor of 4. This run-time increase becomes a problem as the geometric complexity of simulations grows and more detailed meshes are used.

To ensure that the search routines do not dominate run-time, it is important to split the work across MPI ranks. However, splitting this process across ranks will only decrease the search run-time by a linear factor, since the nodes in an interface will still have to search every node in the other interface. This is because it is not known which cells correlate to each other and so there is no way to restrict the search area.

Instead, the interfaces can be split so that the domain over which each node has to search is restricted, highlighted in the second example in Figure 2.8. This requires knowledge as to how far the sliding interface will move, as all required cells from the other interface must be included in the restricted domain. By reducing the size of the domain, and splitting the work across MPI ranks, the run-time can be decreased by the number of MPI ranks squared in the best case. Consider a future test case involving geometrically complex meshes, 10 times finer than the existing production case. In a sliding plane example, this will increase the search routine run-time by a factor of 100 (assuming the interface size has increased by the same factor as the mesh). If manual partitioning is used, and partitions are sub-divided by a factor of 10, increasing coupling resources (MPI ranks) by a factor of 10 should result in little-to-no increase in coupling run-time compared to the existing mesh. If no partitioning is used, then the number of MPI ranks has to be increased by a factor of 100 to achieve the same effect.

It is therefore advantageous to partition the mesh to ensure maximum parallel efficiency in sliding plane problems. However, since the boundaries in the rotor/stator test cases are radial, they cannot be split during run-time and must be done manually. It makes the process time consuming, particularly when determining how to best split the MPI ranks between the main applications and the coupling, as this process must be done via trial and error. Most significantly, no existing coupling library has tools to help with this problem.
Figure 2.8: Allocating multiple ranks per coupler unit divides up the search, but each cell must search the entire opposing boundary (a). By manually partitioning the interface, the search space on the opposite boundary can be restricted (b). Here, the interface search routines have been split across 4 MPI ranks in total, shown by the different colors in each interface.

The brute force search method can be replaced with an Alternating Digital Tree (ADT) method [Bonet and Peraire, 1991], an implementation of a binary search tree which results in search run-time of $O(N \log(N))$ rather than $O(N^2)$, where $N$ is the size of the interface. While this can provide a significant reduction in search time, this is only true if the number of partitions are limited. If the domain can be infinitely partitioned into small enough sections, each with their own CU, there will be little difference between the two approaches. Of course, this is not practical in real-world coupling scenarios, and the motivation to use a tree-based approach will be explored later in this thesis. Importantly however, as with the brute force search, the partitions must still be split manually by hand before run-time to ensure maximum efficiency.

To summarize, the RR Coupler has 3 methods of increasing parallel efficiency within the coupling process. The first is to assign multiple MPI ranks to a single CU, which splits the search workload between ranks but does not partition the opposite interface. Figure 2.8a provides an example of the setup. A single coupler unit has been allocated two MPI ranks. These MPI ranks are shared between Interface 1, shown in red, and Interface 2, shown in blue. The solid cells are allocated one MPI rank and the shaded
cells are allocated the other MPI rank. Although the interface has been divided, and the computation of each segment can run in parallel, each segment will still need to search through every single cell in the opposite interface. In Figure 2.8a, the first MPI rank of the coupler unit (solid cells) is searching for overlap between the first cell of Interface 1 and the entirety of Interface 2. Each coupler unit is given its own slice of the opposite search space, but since we are only using a single coupler unit, its partition is the same size as the whole of the opposite interface.

The second approach is to manually partition the mesh, assigning each part to a CU, and having multiple CUs between each DS. A single MPI rank can then be assigned per CU. This is an implementation of the second example in Figure 2.8b. Here, the solid and shaded cells are allocated their own coupler unit, each with a single MPI rank. The user manually splits each interface in two, such that when the coupler unit searches for overlap between the first cell of Interface 1 and all of Interface 2, its search area is restricted to only a subset of Interface 2. This is shown in Figure 2.8b as a red box around a subset of Interface 2’s cells.

Finally, a hybrid approach can be used, assigning multiple CUs and multiple MPI ranks per CU. All 3 of these approaches will speed up the interpolation equally; however from a performance perspective, the second approach is the best, as the reduction in search time is the greatest when using this method.

The issue with the second option is that unlike with rank assignment, which can be done by simply changing the run-time command, partitioning the interfaces must be done manually. This is time consuming because the partitions must be a similar size to one another to ensure load balancing. In addition, as the number of partitions (and CUs) increase, the likelihood of one interface having no mapping to cells in the opposite interface also increases, which causes the RR Coupler to crash. As a result, having insight on an optimized CU, DS and PS configuration can significantly reduce overall setup time.
2.4 Mini-apps and a mini-simulation

2.4.1 Mini-apps

As hardware and software advancements continue to evolve, and the complexities of scientific codes grow, it becomes challenging to evaluate new programming models and optimizations at the scale that is required using production applications [Owenson et al., 2020]. Instead, small but representative ‘proxy’ or ‘mini-apps’ can be used, which are simplified versions of larger codes which capture the compute patterns and performance characteristics of production applications. The need for mini-apps stems from the fact that as high-performance computing applications grow in scope, small benchmarks such as LINPACK no longer accurately capture the characteristics of production workloads [Reguly et al., 2015].

The use of mini-apps is widespread, and a variety of mini-app collections exist which provide representative proxies of scientific computing applications. One example is the UK Mini-App Consortium [UKMAC, Accessed September 2022], which includes CloverLeaf, a hydrodynamics application which solves the Euler equations using an explicit second order method [Mallinson et al., 2013], and TeaLeaf, which solves the linear heat equation using implicit solvers [McIntosh-Smith et al., 2017]. Another suite is the Mantevo project [Mantevo, Accessed September 2022], consisting of a number of mini-apps which model many common scientific computing domains. This includes solving PDEs on structured and unstructured grids using both implicit and explicit methods, as well as hydrodynamics, molecular dynamics, and circuit simulation codes. Finally, there is the Exascale Computing Project (ECP) Proxy Apps Suite [ECP, Accessed September 2022], which includes a variety of proxy applications modeling molecular dynamics, particle transport, and quantum chemistry.

As the concurrent execution of multiple physical models becomes standard practice, the requirement for proxy tools to model these coupled simulations grows in importance. The complexities of such simulations add additional challenges when evaluating optimizations and new programming models on applications, as the performance of a coupled system may not match that of an optimized application due to bottlenecks elsewhere in the simulation. This necessitates the need for a coupler mini-app, modeling the design
and characteristics of a production coupling framework while making simplifications to enable rapid design space and run-time setup exploration using mini-apps. The CPX mini-coupler presented in this work is the world’s first coupler mini-app [Powell et al., 2021], and allows fast development, execution and evaluation of coupled proxy multi-physics simulations.

2.4.2 Building a mini-simulation

To build a ‘proxy’ coupled gas-turbine engine simulation, three proxy or mini-apps are required: a mini-coupler, a standalone CFD mini-app, and a standalone particle mini-app. Using a combination of the mini-coupler and the CFD mini-app, the compressor or turbine stages can be modeled. By adding the particle mini-app between these two stages and connecting it using the mini-coupler, the simulation can be extended to a proxy coupled gas-turbine engine simulation. Importantly, although we refer to the simulation as a full-engine simulation, we are limiting our scope to only the interaction between fluids, omitting any structural component simulation; the current state-of-the-art simulations, discussed in Section 2.3.2, are made up of at most the 3 core engine stages (HPC, Combustor, HPT). Thus, we consider such additions to be future work, detailed in Section 6.2.3. While the design of the mini-coupler component will be detailed in further chapters, we will discuss the other mini-apps that we will use in our proxy simulations.

2.4.3 MG-CFD - A density solver proxy

For the proxy CFD application we chose an existing mini-app, MG-CFD. MG-CFD has previously been demonstrated to be an accurate proxy for predicting run-times and scaling behavior of the production density solver CFD code used by Rolls-Royce [Owenson et al., 2020].

MG-CFD is a three-dimensional finite-volume discretization of the Euler equations for inviscid, compressible flow over an unstructured grid, and is based on the CFD application by Corrigan et al, included in the Rodinia benchmark suite [Che et al., 2009]. The version used in this work is written using the OP2 DSL [Mudalige et al., 2012], which enables the automatic generation of parallelizations targeting clusters of multi-core CPUs and GPUs. This means that MPI, OpenMP and SYCL versions of MG-CFD can easily be built, as
well as CUDA and HIP binaries that can run on Nvidia and AMD GPUs. Our primary focus is the MPI version, since it is the fastest CPU version of the code on any given platform [Reguly et al., 2021]; however, we will also briefly look at the CUDA version of the code when developing and evaluating performance of a compressor mini-simulation.

This proxy application is designed to be a representation of the Rolls-Royce density solver’s \textit{iflux} and \textit{vflux} loops, which are part of its nonlinear solver and make up nearly half of the run-time. These handle inviscid and viscid flow, with the remainder of time spent in gradient sensitivity and calculating the source term of other equations. MG-CFD predict run-time of the \textit{iflux} loop with less than 16\% error when not using Single Instruction Multiple Data (SIMD) extensions [Owenson et al., 2020]. The \textit{vflux} loop is more computationally intense than \textit{iflux}, but it has a similar memory access pattern, thus the number of cycles can be increased to model \textit{vflux}. Consequently, MG-CFD fulfills our requirement of a proxy application for the density solver, particularly its most time consuming loops \textit{iflux} and \textit{vflux} which determines total run-time.

2.4.4 SIMPIC - A pressure solver proxy

The pressure solver operates using a synchronous coupled Lagrangian-Eulerian approach; within a time-step, flow, combustion, and turbulence fields are updated, passed to the particles, and the particle field is then updated [Thari et al., 2021] [Anand et al., 2013]. This process is then repeated for the next time-step, in Figure 2.9.

In contrast to MG-CFD, a suitable functional mini-app for the pressure solver is not available. Hence an existing electrostatic mini-app, SIMPIC [Yee and Hopkins, 2020], developed by Sandia National Laboratories, was selected to represent the pressure solver. SIMPIC, in this case was chosen to act as a ‘performance proxy’ - an application in a similar domain, designed to replicate the performance characteristics of an application rather than the functions within. The aim with a performance proxy is to demonstrate how the parallel efficiency of one solver can affect the scalability of a large connected simulation.

SIMPIC is used to model plasma interaction within nuclear fusion simulations, combining a particle-in-cell approach with a Poisson solver. Similar to production pressure solvers, it follows a synchronous Lagrangian-Eulerian approach; each time-step, the pro-
Figure 2.9: A comparison between the compute-communication pattern of both the pressure-solver and SIMPIC [Houzeaux et al., 2016] [Thari et al., 2021] [SIMPIC, Accessed May 2022]

gram solves an electric field, passing this information to the particles, which then update their field respectively [SIMPIC, Accessed May 2022]. A comparison between the two applications can be seen in Figure 2.9. SIMPIC is small (LoC≈ 1500), open source, and follows our motivating compute-communication pattern found in a typical pressure-solve. It is also written in C++, thus simplifying the integration into the CPX mini-coupler in comparison to say a Fortran-based pressure-solver [Anand et al., 2013]. Additionally, SIMPIC input files are highly customizable, enabling us to generate a selection of test cases which replicate different parallel efficiency characteristics and pressure solver run-times. However, there are limitations in using such an application. For example, SIMPIC operates over a 1-dimensional domain compared to the 3 dimensional domain of a typical combustion pressure solver. Nevertheless, the utility of the application in this research is its performance characteristics and scalability, not the underlying solver. It is for this reason we describe it as a ‘performance proxy’, rather than a direct proxy application.

SIMPIC is used to model plasma interaction within nuclear fusion simulations, combining a particle-in-cell approach with a Poisson solver. Similar to production pressure solvers, it follows a synchronous Lagrangian-Eulerian approach; each time-step, the program solves an electric field, passing this information to the particles, which then update
Figure 2.10: A breakdown of the 1.5Bn Rolls-Royce test case setup for both the mini-app and full-scale simulation (Right). The chambers have been labeled on a representative RR Trent XWB Engine [Rolls-Royce, Accessed May 2022] (Left, reproduced with permission).

their field respectively [SIMPIC, Accessed May 2022]. A comparison between the two applications can be seen in Figure 2.9. SIMPIC is small (Lines of Code (LoC) \( \approx 1500 \)), open source, and follows our motivating compute-communication pattern found in a typical pressure-solve. It is also written in C++, thus simplifying the integration into the CPX mini-coupler in comparison to say a Fortran-based pressure-solver [Anand et al., 2013]. Additionally, SIMPIC input files are highly customizable, enabling us to generate a selection of test cases which replicate different parallel efficiency characteristics and pressure solver run-times. However, there are limitations in using such an application. For example, SIMPIC operates over a 1-dimensional domain compared to the 3 dimensional domain of a typical combustion pressure solver. Nevertheless, the utility of the application in this research is its performance characteristics and scalability, not the underlying solver. It is for this reason we describe it as a ‘performance proxy’, rather than a direct proxy application.

2.4.5 Mini-simulation setup

In this thesis we will focus on two representative test cases from Rolls-Royce, one a large compressor model of 5Bn cells, and smaller full-engine test case of 1.5Bn cells. Figure 2.10 shows this latter setup of the complete engine mini-app simulation, both in relation to the production codes in the full-scale simulation, and the physical engine itself. The simulation models the last blade row of the Intermediate-Pressure Compressor (IPC), shown in blue, the High-Pressure Compressor (HPCp), in orange, the Combustor, shown
in red, and finally the High-Pressure Turbine (HPT), shown in yellow. Note that in the large compressor test case we will examine first, this consists of only the high pressure compressor and 10 blade rows rather than 12, albeit at a larger scale.

For every blade row within the simulation, there is a dedicated instance of a solver allocated, with the solver instances being connected by coupler units, which are allocated by the coupling framework. The structure of the mini-app simulation is the same as the full-scale simulation with the production Rolls-Royce code. The instances of the density solver are replaced by MG-CFD, the instance of the pressure solver is replaced by SIMPIC, and the RR coupling framework is replaced by the CPX mini-coupler.

2.5 Performance modeling

The use of performance models is well documented, and have been used for a variety of Single Program Multiple Data (SPMD) applications. These include models for particle transport and wave-front applications [Mudalige et al., 2008] and nonlinear solvers [Bunt et al., 2016] [Bunt et al., 2013]. Performance models incorporate a number of parameters, such as domain size and frequency of certain loops, using these to produce an approximation for the run-time, or an optimized setup. A full literature review of performance prediction of parallel applications was recently completed by Flores-Contreras et al. [Flores-Contreras et al., 2021]. In that work, performance prediction methods were organized by the domain being modeled, and the technique used to predict the run-time.

The aim of the performance model in this work is to predict the run-time of each of the solver instances, as well as the coupling framework. As we will later explore, this is to determine where MPI ranks should be allocated such that the run-time of the system is as low as possible. As the aim of proxy applications is to reduce the complexities of evaluating and optimize coupled simulations, we will aim to choose methods which are as simple as possible while delivering good accuracy.

For modeling solver run-time, we will use a curve-fitting approach that is application independent, so it can quickly and easily be applied if new solvers or optimized solvers are added to the simulation. The method can be considered a ‘characterization’ prediction approach, which is based on application execution time across test cases and does not involve code instrumentation. For modeling the run-time of the coupler, we will use an
application dependent approach, examining the different coupling components as a function of input size and compute resources. This can also be considered a ‘characterization’ prediction approach, specifically a hybrid profile method. These are combined with an iterative algorithm which is used to allocate the MPI ranks.

2.6 Summary

In this section, we have examined the need for code coupling and the different approaches that can be taken to achieve it, including their requirements and limitations. A breakdown of how coupling is used in a coupled gas-turbine compressor and full engine was detailed, and an overview of the Rolls-Royce coupler framework used to build these simulations was provided, along with its limitations. The role of mini-apps, both in general and in this use case was laid out, with the specific tools and their relevance to production applications examined. Finally, we presented a brief description of performance modeling, and categorized our prediction efforts in relation to existing work.

Although achieving virtual certification of gas-turbine engines is still in its conceptual phase, with only a single example of a successful 3-component \cite{Arroyo et al., 2021} simulation being run, already there exists significant challenges which need to be addressed to achieve efficient simulations. These include:

- Different mesh sizes of each solver instance which requires ranks to be allocated such that no instance slows down the simulation (Chapter 5.1)
- Different interface sizes, which also need to be allocated the correct number of ranks to not slow down the system (Chapter 5.1)
- Different interface types, which have varying amounts of each stage of the coupling process (Chapter 5.1)
- Different timestep lengths and iteration count of solvers due to the design, e.g implicit vs explicit methods (Chapter 5)
- Different application performance characteristics, such as if one solver is bottlenecked by communication (Chapter 5.3)
• Different coupling sub-techniques, such as optimizations to the search or interpolation routines (Chapter 4.4)

The purpose of this work is to create a suite of tools that aid in developing solutions to these issues for a coupled gas-turbine engine simulation particularly considering simulation on large-scale massively parallel HPC systems. It aims to provide a blueprint which can be extended and improved for future, more complex simulations.
Chapter 3

Building a Mini-Coupler and Compressor Mini-Simulation

Having established the required solver mini-apps, we will start by first looking at a compressor simulation. To do so, we will lay out the organization of the mini-coupler and its associated tools, providing a comparison between the mini-simulation and the production-grade compressor problem of interest. Section 3.1 details the design of the mini-coupler, covering the initial design considerations as well as the components and computation of both the RR Coupler and CPX. Section 3.2 builds the empirical performance model, used to predict the parallel efficiency and run-time of the mini-apps. Section 3.3 establishes the resource allocation algorithm which integrates the model, with Section 3.4 recapping the design philosophy. Section 3.5 details the production-grade problem and the mini-app equivalent, with Section 3.6 summarizing the chapter.

3.1 Building the CPX mini-coupler

3.1.1 Initial design considerations of a mini-coupler

When developing a simplified version of a large code, it is important to determine which parts of the larger application’s design and logic should be included and which can be streamlined or removed. The central aim of any proxy or mini-application is to have sim-
ilar performance and scalability characteristics to the larger code, even if the underlying routines are different [Messer et al., 2015]. The application should be limited in scope, to retain the properties that make proxy applications beneficial, but still close enough to the full application to remain useful. As a result, a number of simplifications were made to the design of the proxy framework in comparison to the full framework:

- The proxy code should use identical, static meshes compared to the different mesh sizes of the full simulation. The meshes should be approximately the size of the average mesh size in the full simulation.

- The interfaces should also have a fixed size, which is not the case with the full framework setup.

- The coupling routines (communication, interpolation and searching) should represent/match the computational intensity of the full simulation. There is no requirement to carry out a valid calculation, and data should simply be discarded once it has been sent back to each MG-CFD Session.

- The framework should support either the RR Coupler’s ‘single Coupler Unit with multiple ranks’ or the RR Coupler’s ‘multiple Coupler Units’ from a performance perspective, switchable with a flag. Since increasing Coupler Units is always preferable, there is no need to have a mixture of Coupler Unit + MPI rank behavior.

- When using multiple coupler units in the RR coupler, interfaces must be split manually prior to run-time due to the fact the interfaces are radial. As the mini-coupler interfaces are fixed sized slices of the larger mesh and mathematical accuracy is not a concern, the interfaces will be split at run-time.

The automatic splitting of the interfaces is the most important design consideration of the mini-coupler, which is discussed further in Section 3.1.2. As the manual partitioning required by the production code takes significant time, testing a variety of coupler unit configurations manually is impractical, yet crucial to achieve efficient simulations of a gas-turbine engine compressor. By addressing the issue in the mini-coupler, different resource allocations can be quickly evaluated enabling full-scale simulations to be run at scale.
Figure 3.1: A typical CPX input configuration file (left), specifying three MG-CFD Sessions, each with 200 MPI ranks, and two Coupler Units, each with 20 MPI ranks. The generated configuration (right), with Multi-Unit Mode (MUM) enabled.

3.1.2 Components of the RR Coupler and design of CPX

Input files

The basic design of the proxy coupling framework, CPX, is very similar to the full scale RR Coupler framework [Amirante et al., 2021b], using MPI for distributed-memory parallelization. The input is read in from an input configuration file which specifies the number of MPI ranks assigned to each Coupler Unit, the number of MPI ranks assigned to MG-CFD Sessions and which Coupler Units are connected to which MG-CFD Sessions.

A flag can be set in the separate CPX global configuration file, known as Multi-Unit Mode (MUM), which switches between running a Coupler Unit in a ‘1 Unit per rank’ mode and a ‘Single Unit with many ranks’ mode. When enabled, the interface is split across ranks during the CPX brute force search routine. When disabled, each rank must perform a brute force search over the entire interface. As a result, the scaling behavior in these two scenarios can be tested quickly without having to manually divide up the mesh, since the CPX meshes are static and the interface is predefined. Due to its rotating domain, using the RR Coupler would be significantly more time consuming. Note that CPX can only
struct unit{
    char type; //either M for MG-CFD or C for Coupler unit
    int processes;
    char coupling_type; //either S for sliding plane or C for CHT
    std::vector< std::vector<int> > mgcfd_ranks; //CUs have 2 MG-CFD instances, solver units 1
    std::vector< std::vector<int> > coupler_ranks; //MG-CFD instances can have multiple couplers
    std::vector<int> mgcfd_units; //the 2 MG-CFD units coupled by this coupler unit
};

struct locators{ //one locator per rank - states type and relative position
    char typelocator; //either M for MG-CFD or C for Coupler unit
    int placelocator; //e.g 1 for 1st Coupler/MG-CFD unit, 2 for 2nd Coupler/MG-CFD unit
};

Figure 3.2: The two structures which contain coupling information in CPX

ever have 1 physical Coupler Unit per 2 MG-CFD Sessions (i.e only 1 communicator), but enabling MUM, configures the single Coupler Unit to scale in a similar manner to that of using multiple RR Coupler CUs.

Figure 3.1 shows an example CPX input configuration file, along with the resultant CPX configuration with MUM enabled. The first line specifies the total number of units in the simulation, in this case 5 (3 MG-CFD units and 2 CUs). Lines 3-5 specify the type of solver instance, which here is made up of instances of MG-CFD, with 200 ranks being allocated to each. Lines 7-9 are used to specify the parameters of the first coupler unit, assigning it 20 ranks and instructing it to couple solver instances 1 and 2. Lines 11-13 are used to specify coupling between solver instance 2 and 3, again with 20 ranks assigned to the coupler unit.

The CPX input file is read when the application starts, similar to that of the RR Coupler in Figure 2.6. To input the MG-CFD mesh file, a parameter is passed on the command line at run-time. The parameter is then passed to the main method of each of the MG-CFD instances which read the mesh input file. This process is the same as the standalone application, except with the extra step of being passed through the mini-coupler first being read by MG-CFD.

Mini-coupler setup stages
The MPI Commworld communicator must be split up into smaller sub-communicators (shown in blue) for each solver or coupler unit instance to handle their internal communication.

When the CPX binary is ran and the input file is opened, the first job is to allocate the cores specified in the input file and lay out the structure of the simulation. This must be done in such a way that any MPI rank allocated to an instance of a solver or coupler unit can at any time find out which units they are connected to, and which ranks they are sending and receiving information from.

To do so, we make use of two data structures that can be seen in Figure 3.2. The first data structure, named ‘unit’, stores the type of a unit (either coupler or MG-CFD), the number of ranks allocated to that unit, the coupling type (which will be sliding for all units in a compressor), and a list of ranks for itself and any units its connected to. If the unit is a coupler, it also contains the identifier of the solver units it is coupling. The second data structure, ‘locators’, stores the unit type and whether the unit is the 1st or 2nd unit attached to a particular coupler unit.

CPX builds two Array-of-Structs (AoS) using these two structures. One array contains a ‘unit’ struct for each instance of a solver or coupler unit in the simulation. The other contains a ‘locators’ struct for each MPI rank in the simulation. Using the locators AoS,
any MPI rank can determine the type of unit it belongs to, and the identifier of the unit it belongs to (e.g. 3rd MG-CFD unit or 2nd Coupler unit). The unit type and identifier can then be cross referenced with the units AoS and find its rank in either the mgcfd_ranks or coupler_ranks vectors. By using these two AoSs, any rank can find out its exact location and role in the coupler simulation.

The next task to be done is to create a new MPI communicator for each Coupler or MG-CFD unit, such that the communication performed internally by the instance is not subject to interference from other units. The idea can be seen in Figure 3.3, which in CPX is done in stages. First, an array of communicators is created, one for each CU or solver instance in the ‘unit’ AoS. The ranks that belong in each communicator are then distributed to MPI groups, also one for each coupler or solver instance, using the information stored in the structures shown in Figure 3.2. These groups are then attached to the communicators stored in the communicators array, and each rank then scans through the array to find which new communicator it belongs in. Note that because CPX interfaces are static, multiple coupler units in the production code can be simulated using a single coupler unit, as noted earlier in Section 3.1.2. As a result, in CPX there is only ever one CU between instances of a solver.

Finally, the ‘unit’ and ‘locators’ AoS are passed to the main methods of the solvers, or the coupler units, as well as the communicator that has been assigned to that unit and the instance number (e.g. 2nd MG-CFD unit). The result allows any rank to not only know which ranks to communicate with during computation within an instance, but also which ranks to communicate with when information exchange via a coupler unit is required.

**Computation within RR Coupler and CPX**

During the RR Coupler sliding-plane execution, there are three main operations: (1) Communication - the Density Solver Sessions send data to their Coupler Unit(s), which then sends data back, (2) Search - the Coupler Units must do a brute force search on each of the interfaces to see which nodes overlap between them and (3) Interpolation - the data must be interpolated before being sent back to the Density Solver Sessions.

Out of the above, the search routine is the most expensive, since it is a brute force search
Algorithm 1: The RR Coupler sequence of operations for a Coupler Unit

\begin{algorithm}
\begin{algorithmic}
  \While{true} \Comment{For every time step}
    \State Open communications to receive data;
    \State Search algorithm;
    \For{$i = 1, \text{niter}$}
      \If{$i \neq 1$} \Comment{open communications to receive data;}
        \State Wait until receive is completed;
        \State Interpolate data on target nodes;
        \State Send interpolated data;
      \EndIf
    \EndFor
  \EndWhile
\end{algorithmic}
\end{algorithm}

Algorithm 2: The CPX sequence of operations for a Coupler Unit (CU)

\begin{algorithm}
\begin{algorithmic}
  \For{$i = 1, \text{number of MG cycles}$}
    \State Open communications to receive data;
    \State Wait until receive is completed;
    \If{$i \% \text{searchfreq}$} \Comment{Search algorithm}
      \State Search algorithm
    \EndIf
    \State Interpolate data on target nodes;
    \State Send interpolated data;
  \EndFor
\end{algorithmic}
\end{algorithm}

Figure 3.4: The CPX Coupler Unit (CU) algorithm in comparison to the RR Coupler Coupler Unit (CU) algorithm. CPX uses linear interpolation, a simplified version of the interpolation scheme used in the RR Coupler [Amirante et al., 2021b].

running in $\mathcal{O}(n^2)$, where $n$ is the size of the interface mesh. However, it only needs to be done every time step and not every iteration, since the interface only changes every time step. The interpolate stage, which averages the data from the interfaces, must be run every time step. It runs in linear time, but significantly more number of times that the brute force search. In a typical RR Coupler + Density Solver run, there is at the very least 10 iterations per time step, and often up to 50. The communication stage must also take place every time step, but it also scales linearly with a smaller contribution to
run-time than the other operations.

Figure 3.4 shows the differences between the RR Coupler and CPX operations within each Coupler Unit. CPX follows a similar model to the RR Coupler, with search, interpolation and send routines, but with a few changes. Firstly, due to the static interfaces MG-CFD deals only with Multi-grid (MG) cycles, not time steps and iterations. As a result, the frequency at which the search routine operates is specified in the CPX global input file. Additionally, CPX uses blocking communication in the transfer between each MG-CFD Session and the respective Coupler Unit(s), which must be completed before any computation can be achieved.

In the RR Coupler, the interfaces between each solver instance must be manually partitioned if multiple coupler units are used. In CPX, each interface mesh is automatically created by taking a small slice of the original MG-CFD mesh. As the solvers used in the mini-simulation are not dependent on the results of other solvers, there is no need for the interface data to be be mathematically correct. The automatic interface splitting allows coupler and solver resource configurations to be created and evaluated by simply modifying the CPX input configuration file.

The search routine in CPX is a brute force search, completed by inserting the data from each node at the beginning of a C++ std::vector, and repeating the process for the size of the interface mesh. The result gives the $\mathcal{O}(n^2)$ scaling with mesh size to match RR Coupler’s search algorithm. The interpolation is an average of the interface nodes, which are accessed in constant time. All nodes in the interface are averaged, so the interpolation also matches RR Coupler’s $\mathcal{O}(n)$ scalability. Finally, the process of communication between the MG-CFD Sessions and Coupler Unit(s) is also linear in interface size.

With Multi-Unit Mode enabled, the interface is split between the number of Coupler Unit ranks during the search routine. As shown in Section 3.1.3, if the mesh is split by a factor of $m$, then the run-time of the search algorithm decreases by $\mathcal{O}(m^2)$. The reason is that the interfaces (of both MG-CFD Sessions) will decrease in size by $m$, and the search routine involves a brute force search (of time $\mathcal{O}(m)$) which must be done $m$ times. With MUM disabled, the search routine is repeated $m$ times, to replicate how the run-time of the RR Coupler search algorithm only decreases at most $\mathcal{O}(m)$, where $m$ is the number of ranks, when a single Coupler Unit with multiple ranks is used.
3.1.3 Capturing performance behavior

To ensure that CPX scaling is in line with that of the RR Coupler, we need to ensure that the CPX interpolation and search routines scale in an equivalent manner. We can compute the complexity of the algorithms (big-\(O\)) in these routines as a function of the interface sizes and MPI ranks for both RR Coupler and CPX, to demonstrate equivalence.

Consider two Density Solver Sessions, connected together with one or multiple Coupler Unit(s). Then let \( R_C \) be the number of Coupler Units/ranks, \( N_{LC}^L \) be the interface size of the 'Left' Density Solver Session and \( N_{RC}^R \) be the interface size of the 'Right' Density Solver Session. Then for 1 CU with multiple ranks:

\[
O(\text{search}) = O\left(\frac{N_{LC}^L}{R_C} \cdot \frac{N_{RC}^R}{R_C} + \frac{N_{LC}^R}{R_C} \cdot \frac{N_{RC}^L}{R_C}\right) = O\left(\frac{N_{LC}^L \cdot N_{RC}^R}{R_C}\right)
\]

(3.1)

\[
O(\text{interpolate}) = O\left(\frac{N_{LC}^L}{R_C} + \frac{N_{RC}^R}{R_C}\right) = O\left(\frac{N_{LC}^L + N_{RC}^R}{R_C}\right)
\]

(3.2)

For multiple CUs:

\[
O(\text{search}) = O\left(2 \cdot \frac{N_{LC}^L}{R_C} \cdot \frac{N_{RC}^R}{R_C}\right) = O\left(\frac{N_{LC}^L \cdot N_{RC}^R}{R_C^2}\right)
\]

(3.3)

\[
O(\text{interpolate}) = O\left(\frac{N_{LC}^L}{R_C} + \frac{N_{RC}^R}{R_C}\right) = O\left(\frac{N_{LC}^L + N_{RC}^R}{R_C}\right)
\]

(3.4)

Consider two MG-CFD Sessions, connected together with one Coupler Unit. Let \( R_C \) be the number of Coupler ranks and \( N_C \) be the interface size of each MG-CFD Session (since the interfaces are all the same size). Then for 1 CU with multiple ranks (MUM=OFF):

\[
O(\text{search}) = O\left(2 \cdot \frac{N_C}{2} \cdot \frac{N_C}{R_C} \cdot R_C\right) = O\left(\frac{N_C^2}{R_C}\right)
\]

(3.5)

\[
O(\text{interpolate}) = O\left(2 \cdot \left(\frac{N_C}{R_C} \cdot R_C\right)\right) = O\left(\frac{N_C}{R_C}\right)
\]

(3.6)

For multiple CUs (MUM=ON):

\[
O(\text{search}) = O\left(2 \cdot \frac{N_C}{2 \cdot R_C^2} \cdot \frac{N_C}{R_C} \cdot R_C\right) = O\left(\frac{N_C^2}{R_C^2}\right)
\]

(3.7)

\[
O(\text{interpolate}) = O\left(2 \cdot \frac{N_C}{R_C \cdot R_C} \cdot R_C\right) = O\left(\frac{N_C}{R_C}\right)
\]

(3.8)
The result of the analytical comparison shows that the scaling behavior for the RR Coupler and CPX should be similar, other than the difference of RR Coupler’s left and right Density Solver interface sizes. If the CPX interface sizes are equal in size to the average of the RR Coupler interface sizes, then the scaling should be close. Of course, scaling is affected by a variety of factors, not just the number of ranks and the size of the domain. In addition, since the CPX search routine is artificial and simplified compared to the RR Coupler, even if the search routine’s parallel efficiency is consistent with RR Coupler, its contribution to overall run-time may be different. The difference can be determined via testing however, and thus the comparison shown highlights how CPX scaling should approximately mirror RR Coupler, even if the overall run-time may need to be adjusted.

Although we have shown analytically that the coupling routines should scale similarly to the production code, we also matched the base run-time of the components to align the run-time of the coupled system to that of the real system. For MG-CFD, a 2-stage Density Solver test case was benchmarked across processor counts, with the coupling and setup time subtracted to leave the standalone application run-time. A similar sized test case was then benchmarked in MG-CFD across processor counts, determining that 50 MG cycles is approximately equivalent to 1 Density Solver time-step with 10 Density Solver iterations. For CPX, we separated the search and interpolate routines, benchmarked the mini-coupler and adjusted the number of iterations of the data insertion and data averaging loops to match the production coupler’s performance, with the results shown in Figure 4.2.

3.2 Building the empirical performance model

The aim of the performance model is to predict the run-time and performance characteristics of the coupler units and individual solver instances, in order to give an optimized distribution of cores such that the run-time of the coupled system is as low as possible. When predicting the individual run-times, our aim is to predict the run-time of the mini-apps, rather than the full scale production applications. There are several reasons for the design choice; for one, we aim to use mini-apps where performance and scaling characteristics closely match the production applications, such as MG-CFD [Owenson et al., 2020],
so the predicted run-time from the model should match that of the full-scale simulations using production applications. We will demonstrate this to be the case in later chapters. In addition, optimizations to solvers or memory access patterns are usually integrated in and tested on mini-apps first, so to see the effect of these in a coupled simulation would require modeling the mini-apps anyway. Finally, by focusing on a mini-app simulation using CPX, the core allocation and run-time prediction can be evaluated far more quickly than a simulation using production applications; unlike with the RR Coupler, the mini-coupler interfaces in the compressor and full engine simulation are not radial, so can be instantly divided at run-time rather than spending time manually splicing the domains of each coupler unit.

Although we are predicting the run-times of mini-apps, the goal of the performance model is to predict the run-time and load balance of a coupled simulation using production applications, which we can do so as the mini-apps will be representative of production applications from a performance perspective. Thus, we will need to map the run-time parameters of the Rolls-Royce production density code to the mini-app, such that the user can enter the parameters of the production code (time-steps, iterations per time-step) and it be converted to those of MG-CFD (MG cycles) internally to calculate the run-time of an instance.

Table 3.1: Converting from production to mini-app cycles

<table>
<thead>
<tr>
<th>Production Parameters</th>
<th>Mini-Apps Parameters</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of time steps</td>
<td>MG cycles</td>
<td>$N_{TS}$</td>
</tr>
<tr>
<td>Iterations per time step</td>
<td>MG cycles</td>
<td>$I_{TS}$</td>
</tr>
<tr>
<td>Iterations per time step × time steps × 5</td>
<td>MG cycles</td>
<td>$MG_{cyc}$</td>
</tr>
</tbody>
</table>

Table 3.1 introduces three parameters: the number of density solver time-steps, $N_{TS}$, density solver iterations per time-step, $I_{TS}$, and the number of MG-CFD cycles $N_{TS}$, which is the product of $N_{TS}$ and $I_{TS}$ multiplied by 5. For example, if the density solver is running for 10 time-steps and 10 iterations per time-step, then the run-time of MG-CFD to match the industry parameters would be 500 MG cycles. These symbols allow the conversion from the production code parameters to the mini-app parameters such that the performance model can then begin the resource allocation and run-time prediction.
In our workflow, the performance model is integrated into an algorithm, which is used to allocate the resources depending on the run-time predicted by the model. The run-time prediction can be split into two stages. First, the initial run-time for MG-CFD and the mini-coupler is generated by taking a base test case and scaling the run-time, up or down by size, MG cycle number, as well as the interface size when predicting the initial search and interpolation time in the coupler units. Once done, the empirical model uses linear regression to fit the parallel efficiency curve of MG-CFD and the RR coupler to determine the run-time given a certain number of cores allocated to the instance. We will therefore lay out the parameters required in both of these stages.

### 3.2.1 Initial run-time

<table>
<thead>
<tr>
<th>Production Parameters</th>
<th>Mini-Apps Parameters</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of mesh (millions)</td>
<td>Size of mesh (millions)</td>
<td>( S_{\text{mesh}} )</td>
</tr>
<tr>
<td>Density solver initial run-time</td>
<td>MG-CFD initial run-time</td>
<td>( RT_{\text{main}} )</td>
</tr>
</tbody>
</table>

All initial run-times were ran on the Edinburgh Parallel Computing Centre (EPCC) ARCHER2 Supercomputer [EPCC, Accessed May 2022], which uses nodes of 2x64C AMD EPYC 7742 2.25Ghz microprocessors, with 256GB memory per node. The base run-time of MG-CFD was set by running the NASA Rotor37 8 million cell test case [Denton, 1997] for 25 time-steps, using 100 ARCHER2 cores, giving a run-time of approximately 3.5 seconds. The model therefore divides the number of MG cycles the user inputs by 25, and divide the size of the mesh the user inputs, shown as \( S_{\text{mesh}} \) in Table 3.2, by 8. It then multiplies the base time by these two numbers which gives us the starting time for the MG-CFD instances in the simulation for 100 ARCHER2 ranks. We give the initial time the parameter \( RT_{\text{main}} \), also seen in Table 3.2.

For example, if the user enters in a mesh size of 128M cells, 10 density solver iterations per time-step and 10 time-steps, the model will first convert the density solver time-steps to MG cycles, and then adjust the base case time to the initial run-time of the user’s test case; this process can be seen in Figure 3.5.
\[ N_{TS} = 10, I_{TS} = 10, S_{\text{mesh}} = 128 \]

\[ MG_{cycl} = N_{TS} \times I_{TS} \times 5 = 500 \]

\[ RT_{\text{main}} = \frac{MG_{cycl}}{25} \times \frac{S_{\text{mesh}}}{8} \times 3.5 = \frac{500}{25} \times \frac{128}{8} \times 3.5 = 1120s \]

**Figure 3.5:** The model must first generate an initial run-time for the user’s density solver test case.

It is worth noting that we assume perfect weak scaling, which MG-CFD only achieves up to mesh sizes around 150M cells (for example, a 300M test case achieves 70% weak scaling). However, as we are using the simplification that the rows of the compressor are the same size, this will make only a small difference to the run-time prediction. When examining the entire coupled engine simulation later in this work, where many different mesh sizes are used, we will account for this weak scaling inefficiency when large test cases are used.

After the MG-CFD instances, the model must then determine the initial run-time for the coupler units. The coupler units operate over the interface between domains, which is specified in the input file of the production coupler. Thus, we must add a parameter which is used to specify the size of the interface as a proportion of the overall mesh. The fraction of the mesh rather than the absolute size was chosen as the mini-coupler interface generation uses the same method.

**Table 3.3:** The model must store the size of the interface used in the simulation, calculated from the mesh size and the fraction of the mesh.

<table>
<thead>
<tr>
<th>Production Parameters</th>
<th>Mini-Apps Parameters</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary mesh fraction</td>
<td>Boundary mesh fraction</td>
<td>( B_{frac} )</td>
</tr>
<tr>
<td>Initial RR Coupler interpolation time</td>
<td>Initial CPX interpolation time</td>
<td>( RT_{int} )</td>
</tr>
<tr>
<td>Initial RR Coupler search time</td>
<td>Initial CPX search time</td>
<td>( RT_{sr} )</td>
</tr>
</tbody>
</table>

As discussed in Section 2.3.3, a search routine must be ran by the coupler units every time-step as the radial domain moves and the mapping between interfaces must be
recomputed. In addition, interface data must be transferred to and from and interpolated for every iteration of the solvers. The number of searches and interpolations in the user’s simulation can therefore be determined by the parameters in Table 3.1.

To generate the initial coupling time, we take a similar approach to generating the initial MG-CFD time, taking a base case and scaling up or down depending on the user parameters. Note that since MG-CFD runs more iterations than the production density code, the interpolation routine is called every 10 MG cycles and the search every 50 MG cycles. Our base case for the coupler uses an interface size of 30,000, or 0.4% of the 8M Rotor37 mesh, and was ran for 250 MG cycles using a single ARCHER2 rank. Unlike with MG-CFD, we split the run-time in two: the time taken for the search routines, and the time taken for the interpolation and other coupling functions. The resulting search run-time is 68s and the interpolation time is 12s. The base interpolation time is then multiplied by the ratio between the user mesh size, and the base case mesh size, the ratio between the base interface size and the user interface size, and the ratio between the base number of iterations and the number of iterations the user specifies. We will call this the initial interpolation time $RT_{int}$.

The base search time is also multiplied by these values, however since the search routine is a brute force method, the run-time complexity is quadratic. Thus, we must take the power of these values. We determined a power of 1.8 matches the RR Coupler search scaling as due to other factors such as compiler optimizations and internal communication the run-time does not increase with a perfect quadratic of 2 when interface size doubles. The initial search time, $RT_{sr}$ is added to the initial interpolation time, $RT_{int}$, to get the initial coupler time. The sum of the initial coupler time and the initial MG-CFD time ($RT_{sr} + RT_{int} + RT_{main}$) is the initial run-time of the simulation given 100 ranks per MG-CFD instance and 1 rank per CU. Figure 3.6 extends the example initial MG-CFD run-time prediction in Figure 3.5 to include the complete initial run-time prediction, with the interface size set to be 0.2% of the mesh.
\[ N_{TS} = 10, I_{TS} = 10, S_{mesh} = 128, B_{frac} = 0.2 \]

\[ MG_{cycl} = N_{TS} \times I_{TS} \times 5 = 500 \]

\[ RT_{main} = \frac{MG_{cycl}}{25} \times \frac{S_{mesh}}{8} \times 3.5 = \frac{500}{25} \times \frac{128}{8} \times 3.5 = 1120s \]

\[ RT_{int} = \frac{I_{TS}}{10} \times \frac{S_{mesh}}{8} \times \frac{B_{frac}}{0.4} \times 12 = \frac{10}{10} \times \frac{128}{8} \times \frac{0.2}{0.4} \times 12 = 96s \]

\[ RT_{sr} = \frac{N_{TS}}{10} \times \left( \frac{S_{mesh}}{8} \times \frac{B_{frac}}{0.4} \right)^{1.8} \times 68 = \frac{10}{10} \times \left( \frac{128}{8} \times \frac{0.2}{0.4} \right)^{1.8} \times 68 = 2871s \]

\[ RT_{main} + RT_{int} + RT_{sr} = 4087s \]

**Figure 3.6:** The model generates an initial run-time for the user’s density solver test case as well as the coupling time

### 3.2.2 Fitting application parallel efficiency

Now that the initial run-time of the simulation and each of the components can be determined, the next stage is to predict the parallel efficiency of components such that by using the initial run-time we can predict the run-time of a solver or coupler unit given any amount of cores allocated. We will treat the problem as curve-fitting exercise and use linear regression using two functions. The prediction must also take into consideration the effects of problem size on run-time and parallel efficiency. To do so, we must understand how parallel efficiency of an application changes as compute resources and the problem sizes increase.

We can split a parallel application by its two core components: portions of the code that run serially, and portions of the code that runs in parallel. As compute resources, the run-time of the serial portion will remain the same while the run-time of the parallel portion will reduce [Amdahl, 1967]. Figure 3.7 shows the effect on parallel efficiency when the serial portion of applications is increased.
The parallel efficiency curve of a typical application follows an S shape. When the number of cores is initially increased, the parallel portion of the code makes up almost all of the run-time, so parallel efficiency is close to 100%. As the amount of allocated compute resource grows, the parallel efficiency decreases as the parallel work makes up an increasingly smaller percentage of run-time. Finally, parallel efficiency will tend towards zero as the run-time becomes dominated by the serial part of the code.

To predict the parallel efficiency in the model, we will first benchmark the mini-apps across a number of ARCHER2 nodes. Then, we will create a linear function, interpolating the values between the smallest and largest node allocations. The function will be made up of a constant and a value being subtracted from the constant which grows as the number of cores increases. By doing so the function will fit the parallel efficiency curve of the application. Of course, given that the shape of the parallel efficiency curve is an S, using a linear function will result in error since a curve cannot be modeled by a linear function. In particular, error will significant when an application’s run-time is almost exclusively made up of serial code, resulting in parallel efficiency dropping quickly and remaining very low as the number of nodes allocated increases.

Figure 3.7: The effect on parallel efficiency when the amount of serial code in a parallel application increases.
To mitigate the issue, we will only predict parallel efficiency until it hits 50%. Beyond this point, the reduction in run-time achieved by allocating more resources is minimal. Cutting off parallel efficiency here eliminates the issue of prediction error in codes where parallel efficiency drops quickly, because at the exact point of 50% parallel efficiency the graph will always be linear, as with any sigmoid.

![Graph](image1.png)

**Figure 3.8:** The error of modeling in a highly serial-bound application to 0% P.E, seen in a), with the reduction in error by limiting modeling to 50% P.E shown in b).

The improvement is visualized in Figure 3.8. The blue line is the true parallel efficiency of the application, and the red line is the predicted parallel efficiency. By modeling parallel efficiency down to 0% the error, shown in shaded red, is significant due to the linear nature of our predictive function. However, by only modeling to 50%, the function can closely match the true performance of the application. Note that since the x-axes on parallel efficiency graphs are logarithmic, the linear function appears curved; however when plotted with a linear axis the function will appear as a line.

We have explored how by employing a linear function with some restriction, we can theoretically predict the parallel efficiency of an application instance using the number of cores or compute nodes as a parameter. However, performance modeling which considers serial and parallel portions of code only works when amount of work being done is fixed. Throughout this thesis we will want to model mini-coupler and mini-app performance over a variety of problem sizes, which cannot be modeled using just the approach discussed
so far [Gustafson, 1988]. Our focus has been on strong-scaling, where the problem size is fixed and the amount of compute resources increased. However, as we are using different problem sizes, we must also consider weak scaling, where performance is examined by increasing the problem size at the same rate as compute resources.

To understand how best to model weak scaling, we must consider the characteristics of a typical HPC application. When first increasing the compute resources and problem size, the run-time will stay the same; the serial part of the computation usually does not increase at the same rate as the domain size [Gustafson, 1988], and the application is highly parallel. However, as we are modeling applications that implement explicit multiprocessing, namely MPI, eventually we arrive at a point where the communication time starts to bottleneck the application and the weak-scaling run-time starts to increase [Moríñigo et al., 2019]. Collective operations, which many HPC codes contain [Dichev and Lastovetsky, 2014], are particularly significant when contributing to such bottlenecks.

Our aim with the performance model is by predicting the run-time given a certain number of MPI ranks, we can chose where best to allocate ranks to lower the run-time of a coupled simulation. While in the initial compressor example the domain size of all instances will be the same, it will not be the case when modeling the HPCp-Combustor-HPT problem in later chapters; in these more complex test cases, there is a factor of up to 35 between the smallest and largest domains in the simulation. If we choose to model the weak scaling as a linear function, the model will become increasingly optimistic as mesh size increases, because the slowdown due to communications grows with mesh size. As such, the model would favor allocating ranks to the largest solver instance, even when real-world performance of these large instances may not benefit additional resources being allocated.

As such, we will model the effects of increasing mesh size as a logarithmic function, with the size of the mesh as the input parameter. We incorporate the function into our existing linear function, which will allow us to capture the effects of increasing mesh size. When the size of the mesh is small, increasing mesh size will result in improved parallel efficiency at given number of cores. However, as the mesh size becomes larger, increasing the number mesh size further will have limited to benefit to parallel efficiency. By combining these two functions, we will demonstrate this linear regression approach can predict run-time with a good level of accuracy. The parameters of the model will
be hand picked by benchmarking the codes and choosing parameters which matching the run-time as close as possible.

Since we are only using curve fitting to predict performance, there will naturally be some error, as a ‘top-down’ approach will always be more inaccurate than a ‘bottom-up’ approach which models computation and communication patterns with high fidelity. However, the methods described here are designed to be fast to generate in order to allow researchers to quickly examine the effects of optimizations and different solver components in a coupled simulation. As we will discuss in Section 4.2, the mean prediction error of the methods presented in this work is similar to the prediction error of the existing MG-CFD mini-app, so despite the inaccuracies of the ‘top-down’ approach, the error is not significantly greater than existing tools used for performance analysis of these engine components [Owenson et al., 2020]. Thus, we believe the trade-offs between accuracy and time we have chosen are well suited for the use case of these processes.

| Table 3.4: The two Parallel Efficiency, or ‘scaling factor’ equations for MG-CFD and CPX |
|---------------------------------|---------------------------------|
| **Parameter**                  | **Symbol+Function**             |
| MG-CFD P.E                     | $SF_{\text{main}} = 1 - 0.01 \times \left((\frac{R_{\text{main}}-99}{8})/(\log_e (Scale \times 8 + e))\right)$ |
| CPX P.E                        | $SF_{\text{cpl}} = 1 - 0.01 \times \left((\frac{R_{\text{cpl}}}{8})/(\log_e (Scale \times 8 + e))\right)$ |

In our compressor example, we are modeling the parallel efficiency of MG-CFD and the mini-coupler, thus we will need an function for each. The input of the function is size of the mesh and the number of ranks being allocated, and the output will be the parallel efficiency of the instance. In the model, we refer to the parallel efficiency as ‘scaling factor’ or SF, seen in Table 3.4. The parameters $R_{\text{main}}$ and $R_{\text{cpl}}$ are the ranks allocated to each instance, adjusted as the base MG-CFD case was ran with 100 ranks whereas the base coupler case ran with 1 rank. Scale is the ratio between the base mesh size and the input mesh size. For example, if the user enters 80 million cells for the size of the mesh, Scale = 10, since the base test case is 8 million cells in size. For the CPX mini-coupler, the parameter takes both mesh size and interface proportion into account. The denominator of the $R_{\text{main}}$ and $R_{\text{cpl}}$ parameters was hand chosen by comparing the output functions with benchmarked run-times.
By examining the format of the functions in Table 3.4, it becomes clear how the parallel efficiency is modeled. When the number of ranks is very small, the value is 1, which represents 100%. As the number of ranks grows, the value being subtracted grows in a linearly and performance starts to degrade. As the size of the input grows, the gradient of the function becomes shallower and more ranks can be allocated for a given parallel efficiency, but as this part of the function is logarithmic, the effect is only true up to a point. As mentioned earlier, the rank allocation algorithm which uses the model is only ran until parallel efficiency reaches 50%.

3.2.3 Deriving run-time from parallel efficiency

To derive the run-time of the application or mini-coupler instances, we first must define speedup, obtained by dividing an old run-time with the new run-time. In our case it is the run-time with base number cores (MG-CFD for 100, 1 for the mini-coupler) divided by the run-time with \( n \) cores:

\[
\text{Speedup}(n) = \frac{RT_{\text{old}}}{RT_{\text{new}}} = \frac{RT(\text{base})}{RT(n)}
\]

Next, we define parallel efficiency, which is traditionally the speedup from \( n \) cores divided by \( n \). However, since our base number of cores may be greater than 1 (e.g. 100 for MG-CFD), we must divide \( n \) by the base number of cores \( b \), which will give us how many cores have been allocated as a factor of the base number:

\[
P.E(n) = \frac{\text{Speedup}(n)}{\frac{n}{b}} = \frac{\text{Speedup}(n) \times b}{n}
\]

Substituting the speedup equation into the parallel efficiency equation yields the following:

\[
P.E(n) = \frac{RT(\text{base}) \times b}{RT(n) \times n}
\]

We can then rearrange to get the run-time of the instance for \( n \) cores:

\[
RT(n) = \frac{RT(\text{base}) \times b}{P.E(n) \times n}
\]

Using this equation, we can determine the run-time of each application instance by using the functions in Table 3.4 and the previous run-time. Before any cores have been allocated, the previous run-time is the initial run-time of the base case, calculated using the process
shown in Figures 3.5 and 3.6. For the coupler units, we use the parallel efficiency in Table 3.4 for the interpolation routines and the square of this value for the search routines, obtaining the run-time by taking the sum of these values. Thus, the resource allocation algorithm which applies the model adds 1 to the output of the function, since the square of a value less than 1 will always be less than the original value.

We can see how the process works by using the initial run-time in Figure 3.5, which was 1120 s with 100 MPI ranks allocated. The mesh size was 128 million cells, so Scale = 16 as the mesh is 16 times bigger than the base case of 8 million cells. As we want to find out the run-time if an extra MPI rank is allocated, we first determine the parallel efficiency of the MG-CFD instance with 101 MPI ranks:

\[ SF_{main} = 1 - 0.01 \times \left( \frac{101 - 99}{32} \right) / \left( \log_e (16 \times 8 + e) \right) \]

\[ SF_{main} = 1 - 0.01 \times \left( \frac{2}{32} \right) / \left( \log_e (128 + e) \right) \]

\[ SF_{main} = 1 - 0.000128 \]

\[ SF_{main} = 0.999872 \]

We can then plug in the result parallel efficiency value into the run-time equation we derived earlier:

\[ RT(101) = \frac{1120 \times 100}{0.999872 \times 101} = 1109.1s \]

Thus, by allocating an extra MPI rank, the predicted run-time falls from 1120 seconds to 1109.1 seconds.

### 3.3 Applying the empirical performance model

In Section 3.2, we saw how the performance model can generate run-time predictions for components of a coupled simulation. Our end goal is for the user to enter the configuration details of the simulation and for us to generate a resource allocation and run-time prediction for the simulation, such that the combined run-time of mini-apps and coupler units is the lowest possible. To do so, we must develop an algorithm which uses the model
to determine which instances ranks should be allocated to, and predict the run-time of the coupled system using the individual run-times.

Algorithm 3: Distribute ranks to coupled MG-CFD instances

**Input:** $T_{mg}$, $T_{CU}$, $Size_{mg}$, $Size_{CU}$, $Iter_{mg, CU}$

**Result:** Optimized core distribution and predicted run-time of coupled simulation

$$
T_{mg} = T_{mg} \ast \frac{Size_{mg}}{Base\_size} \ast \frac{Iter_{mg}}{Base\_iter};
$$

$$
T_{CU} = T_{CU} \ast \frac{Size_{CU}}{Base\_size} \ast \frac{Iter_{CU}}{Base\_iter};
$$

$$
ranks \leftarrow \# \text{ cores};
$$

**while** ranks $> 0$

$$
m_{diff} \leftarrow T(cores[mg]) - T(cores[mg] + 1);
$$

$$
CU_{diff} \leftarrow T(cores[CU]) - T(cores[CU] + 1);
$$

**if** $CU_{diff} > mg_{diff}$ **then**

$$
cores[CU] \leftarrow cores[CU] + 1
$$

$$
subval \leftarrow \# \text{ CUs}
$$

**else**

$$
cores[mg] \leftarrow cores[mg] + 1
$$

$$
subval \leftarrow \# \text{ MG-CFD units}
$$

**end**

$$
ranks = ranks - subval
$$

**end**

$$
output \leftarrow T(cores[mg]) + T(cores[CU]);
$$

$$
output \leftarrow cores[apps], cores[CUs]
$$

The role of the algorithm is to solve a core allocation problem; given a fixed core budget, it must compare the run-times of the MG-CFD and mini-coupler instances, and iterate down through the core budget, allocating cores to whichever delivers the greatest reduction in run-time. Since in our compressor simulation the mesh sizes are the same across each MG-CFD instance, the choice is to allocate ranks to either all MG-CFD instances or all coupler units.

Algorithm 3 presents the iterative algorithm to solve the problem, which we implemented in Python. The base times of the MG-CFD units and the coupler units are first calculated using methods described in Section 3.2.1. For each iteration, the algorithm compares the reduction in run-time of allocating a core to either each of the MG-CFD units, or to the coupler units, choosing the option where the reduction in run-time is the greatest. These
run-times are calculated using the methods described in Section 3.2.3. As the run-time of the simulation is the sum of both, it results in the greatest reduction of run-time for every core allocated. If the parallel efficiency of MG-CFD or the coupler units drop below 50%, then the cores are automatically allocated to the other instance for each remaining iteration. If all instances reach a parallel efficiency below 50%, then the iterative loop will break early and output the current state of the allocation and run-time prediction.

As the number of requested cores may not be a perfect multiple of coupler units and MG-CFD units, the total number of allocated ranks may be slightly above the requested amount. However, as the number of instances in a coupled simulation will be small (typically less than 20) and the number of ranks being allocated are typically large (5000+), we do not consider this to be an issue. If a strict core budget is required, then a single rank can be removed from each of the coupler units or MG-CFD units which will have little effect on performance.

### 3.3.1 Iteration example

Figure 3.9 shows the process of an iteration of the resource allocation algorithm. Here there are three MG-CFD instances coupled together with 2 coupler units. The size of the inputs and the run-times are not realistic, and are chosen just to illustrate the steps performed in each iteration. The MPI ranks of each instance are displayed above, and the individual run-time of each instance is displayed below. In the iteration of the algorithm shown, a core is theoretically allocated to each (shown in yellow) and the new run-time of each instance is computed. The run-time prediction is generated using the previously mentioned methods. In the example, the allocation to the coupler units reduces their run-time by 11 seconds, compared to 1 second to the MG-CFD units. As a result, in this iteration the coupler units are allocated the MPI rank, and only its run-time is updated, shown in the final part of the figure.

It is important to note that since the run-time of the system is the sum of coupler time and the solver time, the rank is allocated not to the instance which has the greatest run-time, but to the instance which delivers the greatest reduction in run-time. Situations may arise where either the solver instance or coupler units may take much longer than the other, but the MPI rank is allocated to the faster unit because the run-time reduction
is lower. Such a scenario is likely to happen if using a solver that has very poor parallel efficiency, or when a significant number of cores have already been allocated, in which case allocating an additional rank is unlikely to significantly reduce run-time.

![Diagram of resource allocation algorithm](image)

**Figure 3.9:** An example of a loop iteration of the resource allocation algorithm.

### 3.3.2 Algorithm inputs and outputs

The input to the algorithm consists of two parts. The first component is a CPX input file, such as in Figure 3.1, which specifies the configuration of the coupled simulation. The
other input is the size of the mesh used for each MG-CFD unit, the number of iterations or time-steps being ran in the simulation, and the size of the interfaces that the coupler units run their routines on. The algorithm produces two outputs: the predicted run-time of the simulation, and the core allocation to each of the instances and coupler units.

### 3.4 Summary of performance model design

![Figure 3.10: A breakdown of how the empirical model is used to generate the resource allocation and predict the run-time of a coupled simulation.](image)

We are applying these techniques to a coupled compressor simulation involving instances of a density-based CFD solver coupled together using sliding planes. However, as previously detailed, there are many multi-physics problems, involving different solver and coupling approaches, which also require these techniques as simulation complexity increases.

Figure 3.10 presents a breakdown of the methods we used. Steps 1 and 2, detailed in Section 3.2, involve benchmarking test cases, hand-picking equation parameters, and creating parallel efficiency functions to predict application performance. Step 3, described in Section 3.3, describes the iterative process of allocation cores and generating output.
3.5 Building the mini-simulation

3.5.1 Production-grade problem

The large scale production-grade compressor problem we are attempting to replicate with proxy tools is the Rig250 setup from Deutsches Zentrum für Luft-und Raumfahrt (DLR), seen in Figure 1.2 and shown again in Figure 3.11. Rig250 is a 4.5 stage test rig, consisting of an inlet guide vane, four rotor-stator stages followed by an outlet guide vane at the rear of the engine, for a total of 10 blade rows [Marciniak et al., 2014].

There are two models of our test case which are the focus of this work [Mudalige et al., 2022]. The first is a coarse geometry, known as ‘Single Passage’, consisting of all 10 blade rows with a size of 430M nodes. We also consider a subset of the model consisting of only the first two blade rows, with a size of approximately 20M nodes. The second uses a finer geometry, known as ‘Full Annulus’, for a total of 4.6Bn nodes across the 10 blade rows.
Similarly, a subset of the model is also available, again consisting of the first two rows and is 653M nodes in size. Each blade row is simulated using Rolls-Royce’s production density solver, connected together using their production coupling framework, the RR Coupler. The full model therefore consists of 10 density solver instances, with the subset consisting of 2 density solver instance, connected with coupler units between each instance.

### 3.5.2 Mini-app equivalent

We will model these test cases by using 10 instances of MG-CFD for the full rig, and 2 instances of MG-CFD for the subset, coupled together with the CPX mini-coupler. MG-CFD will be solving instances of the publicly available NASA Rotor37 mesh [Denton, 1997], which represents the geometry of a transonic axial compressor rotor, widely used for validation in CFD. The geometry is available in a variety of discrete detail levels, from 1M nodes to 300M nodes. For the coarse subset modeling, we will use a mesh of 8M nodes, and for the finer geometry we will use a mesh of 300M nodes. Importantly, none of the software stack and input files have strict licensing restrictions, so unlike the industry codes, these tools can be ran on any system without Intellectual Property (IP) issues.

### 3.6 Summary

In this chapter, we detailed the design and operation of a mini-coupler application, which to our knowledge is the first coupler mini-app to be developed. Alongside the mini-coupler, we presented an empirical performance model, which when combined together provide a suite of tools designed to assist in the increasingly challenging task of resource allocation in coupled gas-turbine engine compressor simulations. In addition, we have detailed our production-grade test case of interest and the setup and configuration for its replication in the proxy realm. We believe the techniques presented are well-suited for other coupled multi-physics simulations, where resource allocation by brute-force is not possible at current and future problem sizes. In addition, the ability to run coupled mini-simulations and the performance model on any hardware platform allows for additional information gain over running mini-apps in isolation, useful for evaluating systems for acquisition.
Chapter 4

Performance Analysis and Prediction of Compressor Test Cases

The use of a mini-simulation and empirical performance model allows domain scientists the ability to determine the core allocation and performance of the system and different components without the use of production applications or settings. However, this is only true if the model and mini-apps can deliver accurate performance predictions. Thus, in this chapter we explore the accuracy of the mini-simulations and the performance model. Section 4.1 compares the performance of different allocations of coupler units in the mini-coupler and the industry coupler using a small internal test case. Following this, Sections 4.2 and 4.3 examine performance in an industry-motivated problem, further validating the performance of the mini-app and the resource allocation of the empirical model. Section 4.4 presents a speculative analysis of a large full-scale industry-motivated problem, highlighting the limitations with the existing RR coupler framework. The performance of an updated RR Coupler, along with an updated mini-coupler and predictive model is presented, comparing performance in the large full-scale industry-motivated problem at scale.
4.1 Comparison with an internal RR test case

The first test involves a simple comparison of ~10M cell test cases, with the production codes using internal ~11M average cell test cases and the mini-apps using instances of Rotor37 8M. The test was performed using 2 Density Solver Sessions, with either 100, 200 or 400 MPI ranks each. For the RR Coupler, three Coupler Unit configurations were tested: A single CU with 1 MPI rank, a single CU with 4 MPI ranks and 4 CUs with a single MPI rank. The RR Coupler test case ran for 5 time steps, with 10 iterations per time step, and the size of the boundary mesh was set to be the same across both test cases (~28,000), so the mini-coupler and industry coupler performance can be directly compared.

The equivalent setup in CPX with MG-CFD Sessions was then tested, with 5 search routines, and 250 MG cycles. As previously mentioned in Section 3.1.3, MG cycles being 5x the number of total iterations provides a close comparison. However, as the overall
MG-CFD mesh is smaller than the Density Solver mesh, small differences in the run-time can be expected. When comparing the mini-coupler with the RR Coupler configuration with a single CU with 4 MPI ranks, we set MUM (Multi-Unit Mode) to off, which runs the CPX coupler unit like a single CU with multiple ranks. When comparing against 4 CUs with a single MPI rank, MUM was enabled, changing the behavior of the CPX coupler unit to act like the 4 distinct coupler units in the RR coupler. A comparison between the configurations, as well as an overview of the industry and mini-app simulation can be seen in Figure 4.1. The tests were ran on the 740,000 core ARCHER2 supercomputer [EPCC, Accessed May 2022], which uses nodes of 2x64C AMD EPYC 7742 2.25Ghz microprocessors, with 256GB memory per node. All codes were compiled using the CRAY 10.0.4 compiler and MPICH 8.0.16 MPI.

<table>
<thead>
<tr>
<th>Hydra ranks</th>
<th>CUs</th>
<th>Ranks per CU</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>195.4</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>4</td>
<td>108.1</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>1</td>
<td>77.5</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>1</td>
<td>176.8</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>4</td>
<td>89.0</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>1</td>
<td>59.4</td>
</tr>
<tr>
<td>400</td>
<td>1</td>
<td>1</td>
<td>158.8</td>
</tr>
<tr>
<td>400</td>
<td>1</td>
<td>4</td>
<td>70.5</td>
</tr>
<tr>
<td>400</td>
<td>4</td>
<td>1</td>
<td>42.2</td>
</tr>
</tbody>
</table>

Table 4.1: RR Coupler ~10M Cell Tests (Test Case 1)
Tables 4.1 and 4.2 show the performance comparison between RR Coupler and CPX. The result is displayed in Figure 4.2. The results show that the CPX produces a mean
error of 11% when predicting the production code run-time, with a worse case prediction error of 28%. One noticeable characteristic of the CPX results is that the MUM=OFF results are much closer to the MUM=ON results than for the RR Coupler; this is due to the search routines being simpler on the mini-app than the production code. While they theoretically scale similarly, the search time duration is longer for the production code. This difference accounts for the majority of the prediction error, and since it is always ideal to assign MPI to one CU per rank over assigning multiple MPI ranks to a single CU, this error is largely insignificant. The error when comparing the RR Coupler multiple CU results to CPX (MUM=ON) results is small by comparison. The 11% average prediction error for CPX shows that the CPX mini-app behavior is close to RR Coupler production code in this first test.

### 4.2 Comparison with Rig250 coarse subset

![Figure 4.3: A comparison between the three tested Coupler Unit configurations in the industry simulation and the mini-app simulation running the industry motivated two-row Rig250 coarse subset problem.](image)
The second comparison involves a different \( \sim 10 \text{M} \) cell test case, this time with two rows of the industry motivated coarse DLR Rig250 test case described in Section 3.5.1. Figure 4.3 provides an overview of this test. These meshes have an average size of \( \sim 10 \text{M} \) (slightly smaller than the first test case), but have interfaces approximately 20\% larger in size than the first case. The mini-apps use instances of the same NASA Rotor37 8M test case, but the interface sizes have been increased by 20\%. In this test, two Density Solver sessions are connected together using 1, 2 and 4 CUs, and the number of MPI ranks is increased incrementally from 100 to 800. The Density Solver + RR Coupler test case has 5 time steps and 10 iterations per time step, so the CPX + MG-CFD case is run with 250 MG cycles and 5 search routines. The hardware and software configurations remain the same as the internal test case in Section 4.1.

![Figure 4.4: Rolls-Royce Coupler vs. CPX. 2nd \( \sim 10 \text{M} \) cell test case with 1, 2, and 4 Coupler Units.](image)

Figure 4.4 shows the performance scaling of both CPX and RR Coupler with this test
case. The CPX mini-app produces a mean error of 16% when predicting production code run-time, with the maximum error being 41%. This maximum error is due to multiple MG-CFD sessions becoming out of sync with one another and having to wait for the slowest session whenever data is sent to the coupler. Again, to match solver run-time, we run 5 iterations of the MG-CFD solver for every Density Solver iteration, as a result of MG-CFD’s simplified solver. Due to the similarities in computation and memory access, the two solvers should have similar variance per iteration, but as a result of the increased number of iterations of MG-CFD, it is likely overall variance is higher, resulting in synchronization issues and increased run-time at high numbers of MPI ranks per MG-CFD session. This must be considered when using CPX + MG-CFD as a prediction of RR Coupler + Density Solver run-time. However, when considering that the underlying MG-CFD mini-app has an mean prediction error of 16% [Owenson et al., 2020], the average error from the mini-coupler is small, even with the high core count results, and can be used as a representation of the RR Coupler framework. Furthermore, the overall scalability trend is very similar to that of the RR Coupler, so its qualitative accuracy is high. Thus, if the performance model produces an ‘optimum’ configuration for the RR Coupler, then that configuration should also be optimal for CPX and vice versa.
4.3 Validating the Performance model - Rig250 coarse subset

Figure 4.5: Rolls-Royce Coupler vs. CPX vs. Model Rig250 coarse subset, testing different CU numbers with 400 Density Solver/MG-CFD MPI ranks.

Having examined the CPX mini-coupler, we will next validate the performance model using the same 2-stage industry motivated Rig250 coarse subset problem that was examined in Section 4.2. First, the run-time between the industry simulation, the mini-coupler and the performance model was compared given a fixed number of DS/MG-CFD ranks and an increasing number of coupler units. For this test, the input parameters are set to $S_{mesh} = 10$, $N_{TS} = 5$, $I_{TS} = 10$, $B_{frac} = 0.0033$, with the number of solver ranks set at 400. Figure 4.5 presents the results, with the model predicting the CPX run-time with a maximum error of 29%.

Next, the performance model was challenged to determine the optimum number of CUs are suitable with a core budget of ~850 ranks. For this second test, the input parameters are set to $S_{mesh} = 10$, $N_{TS} = 5$, $I_{TS} = 10$, $B_{frac} = 0.0033$, with $R_{total} \approx 850$. Using this
setup, the model suggests that for 400 ranks for each Density Solver, 64 is the optimum number of CUs, for a total of 864 total ranks.

Figure 4.5 also shows the proposed ‘optimal’ run-time compared to the same setup but with fewer CUs, testing the setup with both the mini-coupler and RR Coupler. For both, the reduction in run-time as a result of increasing CUs decreases the more CUs are added. For this reason, reallocating Density Solver ranks beyond 64 CUs would likely not result in a decreased run-time. When looking at the proposed ‘optimal’ configuration, taking coupler resources and allocating them to Density Solver/MG-CFD sessions also does not result in a decreased run-time. This suggests that the proposed solution is optimal for $R_{total} \approx 850$, and shows that the performance model is working as intended. As the coupling run-time is minimal here, the performance discrepancy between CPX and the RR Coupler run-time is due to the MG-CFD mini-app running faster than the Density Solver.

4.4 Predicting the performance of the full Rig250

4.4.1 Base performance

Using the performance model we can project the optimum configurations for production problems of interest, with CPUs and also GPUs, by converting the GPU run-time into a $RT_{main}$ equivalent, where $RT_{main}$ is the run-time of the Density Solver/MG-CFD on a CPU. We will do this to estimate performance on both in the 10-passage, full annulus Rig250 problem. The total problem size of this simulation is 4.6 Billion cells, where the average mesh size per Density Solver instance is approximately 500M cells. It is worth noting that the model does not take into account any additional overhead for CPU-to-GPU Peripheral Component Interconnect Express (PCIe) communication, which may reduce coupling efficiency when running the coupled Density Solvers on GPUs. As a result, while optimal configurations should be valid, the GPU run-times should be taken with a degree of caution.

Prior testing of standalone the Density Solver code has shown that a node with 4 Nvidia V100 16GB GPUs will produce similar run-time to 550 AMD EPYC 7742 CPU cores. However, given the Thermal Design Power (TDP) of an Nvidia V100 GPU (300W)
The predicted run-times (s) for the 4.6B cell test case on CPU and GPU clusters. The number of Density Solver ranks are fixed at 3700 per DS for the CPU cluster and 40 GPUs per DS for the GPU cluster.

and an AMD EPYC 7742 CPU (225W), when factoring in the power usage of further equipment that’s required for node functionality (such as networking equipment), we estimate that a node containing 4 Nvidia V100 GPUs will consume around 2.5x more power than a node containing 2 AMD EPYC 7742 CPUs. The large instance test case requires approximately 400 GPUs to fit the mesh into GPU memory, which is equivalent (in terms of performance) to 220,000 CPU cores. As a result, the model parameters for the GPUs are set at $S_{mesh} = 500$, $N_{TS} = 10$, $I_{TS} = 10$, $B_{frac} = 0.004$, and $R_{total} = 220000$, producing an optimal configuration of 40 GPUs per Hydra instance, and 1280 CUs per pair. Furthermore, since we estimate that the power consumption of 100 GPU nodes is similar to the power consumption of 250 CPU nodes, using the performance model, we can predict the performance on 250 CPU nodes using 32,000 MPI ranks.

It is worth noting that the CUs must run on the CPU, even when the Density Solver is ran on GPUs, which results in additional power consumption. When this is taken
into account, 100 GPU nodes combined with the CUs is equal to a pure CPU setup of 42,000 MPI ranks. The parameters of the power-equivalent CPU run were therefore set at \( S_{mesh} = 500 \), \( N_{TS} = 10 \), \( I_{TS} = 10 \), \( B_{frac} = 0.004 \), and \( R_{total} = 42000 \). With this setup, the model suggests approximately 3700 ranks per DS and 560 CUs per pair of DSs. The run-time predictions for execution on both 100 GPU nodes and 250 CPU nodes can be seen in Figure 4.6, with the results demonstrating the GPU nodes’ estimated power efficiency benefits over the CPU node configuration for a variety of CU counts.

Similarly, the performance model can be used to predict the optimal configuration as we scale up the number of CPUs or GPUs used for execution. Figure 4.7 shows the run-time predictions for the optimal configurations as the number of computing resources are increased. The model predicts that there is a hard scaling limit (around 10,500 Density Solver ranks per session and \( \sim 600 \) CUs, at which point allocating any more resources does not improve run-time. Internal testing has shown that the Density Solver’s parallel efficiency is only 67% at just 8,200 ranks per session in this test case, so while the predictive

![Figure 4.7: The predicted run-times(s) for the 4.6B cell test on CPU and GPU power equivalent clusters.](image-url)
model may be too aggressive in assuming no speedup beyond 10,500 ranks, any additional speedup will likely be minimal if this is not the case. The coupler scaling is likely limited by the brute force nature of the search routine; the performance model states that even at 600 CUs, 98% of the coupler run-time is made up of the search. The CFD bottleneck is likely memory or cache related, as it is a memory bound code [Reguly et al., 2021].

It is important to note that the number of CUs cannot be increased indefinitely. As the CU number increases, the size of each partition decreases which can cause issues if there is no mapping between a cell in one interface and the opposite interface. Although the stated CU configurations are the optimal setups for these test cases, splitting the interface such that this issue does not occur is difficult. Whilst having larger partitions and a smaller number of CUs will show a slight performance decrease, as shown in Figure 4.6, it will be far easier to set up than the optimal configuration. Any additional time spent during run-time will be offset by the time saved during setup, as the partitions will likely be stable on the first attempt at partitioning. Therefore it may be sensible to allocate fewer CUs than optimal to ensure stability.

Specifically, allocations beyond 64 CUs are not stable enough to run, regardless of the problem size. As seen from Figure 4.5, this is enough to run the coarse Rig250 mesh, but is not sufficient for the finer test case. Figure 4.6 highlights that even using the maximum 64CUs in this problem, the run-time would be dominated by coupling, with 90% of the run-time being from these routines. Since this coupler unit limit is fundamental, changes must instead be made to the coupling routines, most notably switching from a brute force search routine to a more efficient method.

4.4.2 Optimized coupler

Following the conclusions of Section 4.4.1, the RR Coupler’s brute force search algorithm was replaced with a parallel binary tree search [Mudalige et al., 2022]. The updated coupler implements the ADT method [Bonet and Peraire, 1991], a search algorithm designed specifically for determining intersection in geometric spaces. This change reduced the coupling run-time by up to 65% in the Rig250 coarse subset mesh, representing a 2.9x speedup [Mudalige et al., 2022]. Significantly, the time complexity of this search is $O(N\log(N))$, compared to the $O(N^2)$ of the brute force search, where $N$ is the number
of elements. Thus, it now becomes possible to run the fine Rig250 test case.

To replicate the new tree-based search in CPX, the `std::vector` search routine detailed in Section 3.1.2 was modified to use `std::map`, which uses a binary search tree internally. This was benchmarked on a smaller scale, and results were fed back into the performance model. One important note is that from this point onward we will always use CPX with MUM=ON, as all future simulations using industry codes will be ran with 1 Coupler Unit per rank. Using the updated model, we will compare the run-time predictions of the subset and full fine Rig250 test cases with actual results on up to 512 ARCHER2 nodes, or 65,000 AMD EPYC cores. Although the mini-app values have been fed into the model, we will only compare the model to the industry applications. This is because the largest Rotor37 mesh is only 300M nodes in size, whereas Rig250 instances are up to 700M. The production Density Solver results are taken from existing research [Mudalige et al., 2022] in which a more recent DSL (OP2) version was used than we have examined so far, so we have updated the MG-CFD run-time in the model (previously 3.5s) to time taken from a version of the mini-app using this updated DSL (2.1s). For this comparison, we are also using an improved algorithm which supports different sizes meshes for each instance, allowing us to specify the exact size of each component of the Rig250 test case. We will describe these changes in detail in Chapter 5. The comparisons from this point are only in the CPU space, as there is currently no results for the full annulus Rig250 test case on GPUs due to memory limitations [Mudalige et al., 2022].

### 4.4.3 Updated performance model with Rig250 coarse subset

The mesh size of the first two Density Solver instances, as well as the size of the interface handled by the Coupler Units was entered into the model algorithm. Figure 4.8 presents run-time per time-step predictions from the empirical model alongside the results in the the industry simulation using the Rig250 coarse subset mesh, taken from existing research [Mudalige et al., 2022]. As previously mentioned in Section 4.2, this problem consists of 2 Density Solver instances, with each mesh approximately 10M nodes in size. This industry simulation was ran from 15 to 80 ARCHER2 nodes, setting $R_{total}$ from 1,920 to 10,240 MPI ranks. The empirical model was able to predict the simulation with an average error of 20% and a worst-case error of 25%.
The error appears for two reasons; the first is that the model allocates up to 64 coupler units, whereas the simulation using the industry Density Solver is only allocating 30 coupler units. This is because in the mini-simulation, coupler units can be split automatically, whereas in the industry simulation this must be done manually, since splitting the boundary into 64 domains is challenging for a small domain. In addition, since the mini-coupler automatically splits the domain, the coupler units will always be perfectly load-balanced, whereas this is not the case with the production coupler. In the industry results, coupling overhead was between 2-8% whereas the performance model predicts a ~2% coupling overhead. The second is that as previously mentioned in Section 4.2, MG-CFD is not a perfect representation of the production Density Solver, so some degree of error is expected even if the model was 100% accurate.
4.4.4 Updated performance model with full Rig250

After testing the first two rows of the Rig250 coarse subset, we have established that the optimized search algorithm can deliver excellent scaling performance, and that the performance model can accurately predict the run-time in this smaller problem. However, the final test is to evaluate the accuracy of the model in our full motivating problem, the 10-row Rig250 ‘Full Annulus’ compressor test case.

To do so, the details of the full 10-passage test case were first entered into the model algorithm. As this is the full industry-led problem, we chose to compare the model’s run-time prediction for a full revolution of the blade rows, equivalent to $N_{TS} = 2000$. Figure 4.9 presents these run-time predictions from the empirical model alongside the results in the the industry simulation, across 166, 256 and 512 ARCHER2 nodes, or up to 65,536 cores. The mean error is 30%, with a maximum error of 70%. Almost all the error comes from the Density Solver having better strong scaling at 512 nodes than we
predicted using the model, a result of the model predicting mini-app performance rather than performance of production code. The largest publicly available instance we can use in the mini-apps is 300M nodes, but the Rig250 full annulus features meshes with up to 700M nodes. As the model parameters are set on mini-app simulation using publicly available input, it is unknown how well MG-CFD will perform on a mesh this size, so we cannot exactly say whether these results suggest divergence of mini-app weak scaling, or inaccuracies due to model parameters. Testing with large IP protected problems is possible, and depending on the outcome, minor adjustments could be made to the model parameters to lower the error at very large core counts. Due to the compute resources and cost required, it is not something we explored further in this work, but we believe this approach would significantly lower the error seen in the results. If this extreme scale 65,536 core result is removed, then the maximum error becomes just 16% with a mean error of 9%.

Rig250 Full Annulus - Empirical Model vs Industry Simulation - Coupling Time

Figure 4.10: A comparison between the predicted coupling time of the empirical model and the coupling time of the industry simulation in the Rig250 full annulus test case
The final result, shown in Figure 4.10, is the predicted coupling overhead from the model compared to the real overhead from the full annulus simulation at 512 nodes. The model is able to predict the coupling run-time in the full simulation with an error of 6%. Significantly, unlike the results in Figure 4.8, both the model and the industry simulation are now allocating the same number of coupler units in the largest interfaces (64CU), so this is a true like-for-like comparison. The comparison demonstrates in large problems the model can accurately predict this overhead.

4.5 Summary

In the previous chapter, we presented CPX, a mini-coupler designed to match the performance behavior of Rolls-Royce’s production coupler framework. In this chapter, we combined this with the MG-CFD mini-app, and showed that CPX can be used to create proxy configurations that are representative of production simulations carried out with RR Coupler and the RR Density Solver code. We highlighted that the mini-coupler can be used alongside MG-CFD to predict the run-time and scaling behavior of Rolls-Royce’s industry-motivated coupled CFD simulations with high qualitative and quantitative prediction accuracy. Furthermore, we detailed the creation and testing of a performance model that is designed to predict the optimal resource configuration of a given coupled simulation for both CPX and by extension the RR Coupler.

We demonstrated the use of the model to theorize the optimal configuration for a large test case to be simulated on both a 100,000 core cluster and a 400 GPU cluster, comparing power equivalent CPU and GPU setups with the original RR Coupler. From this work, we determined that due to the interface partitioning requirements of the production code, it would not be possible to execute large scale simulations coupled CFD simulations without design changes. The CPX mini-coupler and empirical performance model were adjusted to accommodate a new version of the RR Coupler with an improved search routine. Using this new RR Coupler, we illustrated that we can predict the run-time and scaling behavior of Rolls-Royce’s industry-motivated large-scale coupled CFD simulations with high prediction accuracy.

The only use of production applications was in benchmarking the RR Coupler in order to determine the performance profile for CPX during the design stages, which only had
to be done once. This is a requirement for any mini-app aiming to be representative of a large scale application. Other than this, every mini-simulation result was obtained without any use of production applications or IP-protected test cases; the production results are purely for validation purposes.
Chapter 5

Building and Optimizing a Full Engine Simulation

So far, we have designed and evaluated methods to load balance and predict the performance of a coupled compressor simulation, consisting of coupled instances of CFD solvers. However, the most complex coupling scenarios consist of close interaction of CFD models with other numerical simulations, including FEM and combustion. One such problem extends our previous setup, modeling the full core of a gas turbine engine, consisting of multiple high-pressure compressor and/or turbine stages interacting with the combustion

Figure 5.1: A breakdown of the 1.5Bn Rolls-Royce test case setup for both the mini-app and full-scale simulation (Right). The chambers have been labeled on a representative RR Trent XWB Engine [Rolls-Royce, Accessed May 2022] (Left, reproduced with permission).
chamber. To recap, in this arrangement (see Figure 5.1), cold air entering the engine is compressed before it is delivered to the combustion chamber, where it is sprayed with fuel and ignited. The exhaust air from the resulting combustion provides the thrust that drives the turbines which, in turn, spins the compressor and fan. Currently, the components are designed and manufactured separately, which can lead to reduced engine performance due to optimizations made within individual elements being negated as a result of integration with other components. Without coupled simulations, there is information loss as unsteady interaction between components cannot be modeled [Arroyo et al., 2021]. Such coupled simulations have become furthermore important with the industry ambitions to reach virtual certification of full aero-engine designs [ASiMoV, 2018; European Commission, 2011]. Ultra-high fidelity simulations are required, pushing current model sizes of 10-100 million elements to tens of billions of elements.

The focus of this chapter is on one such complex scenario – the coupled simulation of the compressor-combustor-turbine components, directed by these industry challenges. In this setup, the compressor and turbine are typically modeled using a density-based, explicit CFD solver, with the combustion chamber employing a LES turbulence model with Lagrangian fuel spray [Anand et al., 2013], and a pressure-based, implicit CFD solver. The rotor/stator interaction in the density solver uses the sliding planes method, and with a steady-state approach used to model interaction between the density and pressure solvers [Amirante et al., 2021b]. As with the previous compressor problem, dedicated coupler(s) software implements these interactions. Our motivating problem is a representative 1.5Bn cell test case from Rolls-Royce plc., consisting of 13 compressor rows, a combustor chamber, and two turbine rows. Work is in progress to run this test case using Rolls-Royce’s internal density-based CFD application [Lapworth, 2004], pressure-based combustion CFD application [Anand et al., 2013], and an in-house coupling framework [Amirante et al., 2021a].

When coupling between component simulations, codes are tasked with communicating boundary information, where the coupler(s) map values/fields from one simulation to the other, interpolating data and transferring them as required [Powell et al., 2021]. A key challenge to gain efficient execution is load balancing. Resources (e.g. compute processes) need to be carefully allocated for each component simulation, to elicit maximum concurrency. At the same time, couplers themselves need sufficient dedicated resources to ex-
cute the transfer of information, without which they will lead to performance bottlenecks, slowing down the overall simulation. However, given the different coupling interfaces and interactions between components, the scale of production executions, including long execution times (in the order of days or weeks) and the need to use large HPC systems, a direct, brute force tuning of run-time parameters and load balancing is prohibitively expensive and unachievable under production settings.

As a solution, in this chapter we create a simplified version of a full scale Compressor-Combustor-Turbine simulation using scaled-down, but representative applications. Building on the coupled multi-row compressor mini-simulation from Chapters 3 and 4, with MG-CFD [Owenson et al., 2020] and CPX [Powell et al., 2021] as the proxy for the density-solver and coupler respectively, a new pressure solver proxy modelled by the SIMPIC [SIMPIC, Accessed May 2022] Particle-In-Cell (PIC) code is added to create the compressor-combustor-turbine triple-components as a coupled mini-app simulation. A comparison of the full-scale and proxy simulations can be seen in Figure 5.1, along with a diagram highlighting the related engine components simulated. An extended empirical performance model is developed to reason-about and explore the optimal resource allocation for the coupled mini-apps when executing a large mini-app engine simulation. We then use the analysis and predictions from the mini-app simulation and performance model to predict the optimum allocation of resources for the full-scale simulation of the 1.5B HPC-Combustor-HPT test case.

The primary objective of the work is to predict the best allocation of resources for running coupled CFD/combustion using mini-apps, as there can be a significant bottleneck on the parallel efficiency of the simulation if resources are not adequately distributed. Using the full production codes will waste HPC resources and due to the scale required to run, the problem quickly becomes intractable when reaching current production problem sizes that are at 1B to 5B cells. The challenge of configuration exploration becomes even greater as the amount of compute resources increases. In addition, we aim to determine the bottlenecks for the workload in the codes themselves and examine optimizations, speculating what the optimized run-time would be. Similarly, without mini-apps, it becomes difficult and time consuming to achieve due to the size and complexities of the production codes. Addressing these bottlenecks represents an important step as part of an ongoing push towards virtual certification. While we focus on the behavior of a specific
gas-turbine simulation problem, the use of CPX can be applied when coupling mini-apps in other domains.

Section 5.1 presents the steps that were taken to support a full-engine mini simulation, including integrating the SIMPIC mini-app into the CPX mini-coupler. Section 5.2 details the motivating industry-led problem and the equivalent mini-simulation test case. Section 5.3 determines the SIMPIC input parameters needed to turn the mini-app into a performance proxy for the industry Pressure Solver, and identifies which component of the mini-simulation is likely to be the bottleneck. In Section 5.4 the bottleneck industry application is profiled to understand where the scalability issues lie, and a number of optimizations in wider research are examined. The results of these works are extrapolated and applied to the performance profile of the industry application to estimate the performance of an optimized application. Section 5.5 extends the empirical performance model to include the SIMPIC performance proxy and includes the updated tree-based search within the model, while Section 5.6 details the enhanced iterative algorithm which implements the performance model and can generate a run-time prediction and core distribution in simulations with different mesh sizes, interface sizes and coupling frequencies. Finally, Section 5.7 summarizes the work in this chapter.

5.1 Extending CPX for a full-engine mini-simulation

5.1.1 SIMPIC integration

In contrast to MG-CFD a suitable functional mini-app for the pressure solver is not available. Hence an existing electrostatic mini-app, SIMPIC [Yee and Hopkins, 2020], developed by Sandia National Laboratories, was selected to represent the pressure solver. As detailed in Section 2.4.4, SIMPIC was chosen to act as a ‘performance proxy’ - an application in a similar domain, designed to replicate the performance characteristics of an application rather than the functions within. The aim with a performance proxy is to demonstrate how the parallel efficiency of one solver can affect the scalability of a large connected simulation.

To build a full-engine mini-simulation, the first step was to integrate the SIMPIC mini-app into CPX. Similar to the RR Coupler code, CPX is the starting point of the
simulation, launching instances of the solvers as specified in input files; these solvers are connected by building the solver codes as libraries and linking them at compile time. Thus, SIMPIC was converted from a standalone application to a C++ library by modifying its makefile, resulting in a static library ‘libsimpic.a’. As CPX was started as a branch of MG-CFD, both MG-CFD and CPX are compiled with the same makefile, which builds both in one step. However, as we are now introducing another application type, we must also modify this makefile to include the SIMPIC library. To enable the inclusion of the library, a flag is required to be set, as users may want to run just MG-CFD instances without building and linking SIMPIC. Additionally, SIMPIC was modified such that it works within the MPI communicator passed to it by CPX, rather than initializing its own on startup.

With these changes, CPX was now able to link to the SIMPIC library and launch instances of the solver at run-time. To enable the user to add an instance to their simulation, we next need to modify the routines that read the CPX input configuration file to

**Figure 5.2**: The CPX input configuration file seen in Figure 3.1 (left), specifying three MG-CFD Sessions, each with 200 MPI ranks, and two Coupler Units, each with 20 MPI ranks. To the right is an updated CPX input file with the final unit switched for a SIMPIC instance, along with a declaration of coupling type between each solver instance.
support the inclusion of SIMPIC into the simulations. Figure 5.2 presents the previous input configuration file, seen in Figure 3.1, compared to a new example with changes highlighted in red. The keyword ‘SIMPIC’ can now be used to specify a SIMPIC unit, and users must now specify the coupling type between the units. If the user chooses ‘sliding’, then the search routine and interpolation stages are run at set intervals according to a predetermined ratio which converts the Density Solver time-steps to MG-CFD cycles. With ‘overset’, the interface is much larger, and there is only one search routine at the beginning of the simulation with all other computation being only interpolation at the same set intervals.

Although at this stage the user could include SIMPIC instances in the simulation, they were only linked to a coupler unit unidirectionally. This is to say that while the coupler unit could send information to SIMPIC, the solver was still running in isolation and was not aware of the coupling occurring. To enable bidirectional coupling, we had to include the header file specified in Figure 3.2, which allowed SIMPIC to determine on which rank data should be sent to and received from.

In prior simulations, all solver instances in the simulation were the same type (MG-CFD) and running for the same amount of time-steps. The result of this was the instances were in lock-step and the coupling frequency was the same across each. However, with the introduction of SIMPIC, the number of solver iterations were now different across solvers. To adjust for this, the coupling frequency was made a global parameter which each solver could access during initialization, allowing solvers to convert the frequency to their own time-step units such that they could run at different speeds. With this, SIMPIC was able to be fully integrated and exchange information in a coupled simulation.

5.1.2 Solver input

Due to its origins as a branch of MG-CFD, along with the mesh sizes being similar in the Rig250 test cases described in Section 3.5.1, passing input to the MG-CFD instances was handled the same as the standalone application. At run-time, the user would add an ‘-i’ flag followed by the name of an MG-CFD input file (.dat) as command line arguments to the application binary within their mpirun command or scheduler submission script. This would pass this single input file to all instances of MG-CFD in the simulation, as detailed
in Section 3.1.2. However, as we will explore later, the industry-led test case modeling the entire engine simulation features significant variance between the size of each mesh, so it was not possible to run an accurate simulation by using the same mesh size for all MG-CFD units. The addition of SIMPIC, which like MG-CFD also reads input from command line arguments, further added to the needed to overhaul how solver input is handled within CPX.

SIMPIC

The SIMPIC input method, which previously read parameters from the command line, was re-written to take read parameters from a file called ‘simpic_ parameters’. As previously mentioned, the command-line parameters passed to CPX were by default automatically passed to MG-CFD, so the change was necessary as otherwise SIMPIC would read these instead.

-ppc 60000 -ncpp 38 -nt 90000 -dtfactor 0.000001 -lhsv 20000 -asz 377

Figure 5.3: The SIMPIC input parameters of a typical test case.

Figure 5.3 lists each of these parameters in a typical example. The details of each can be found in SIMPIC documentation [SIMPIC, Accessed May 2022], but we are only interested in the first three; we are using SIMPIC as a performance proxy, and these are the only parameters which affect performance. The first, −ppc, controls the number of particles per cell. The second, −ncpp, controls the number of cells per processor. As a result, if you are performing strong scaling tests a fixed number of cells, this must be divided by the total number of MPI ranks you are using. Finally, −nt is the number of time-steps the simulation is being ran for. By adjusting these parameters, the run-time and scaling characteristics of the code can be modified, allowing us to create test cases that mimic the behavior of the production Pressure Solver code.

The final parameter −asz, highlighted in red in Figure 5.3, is a parameter we have added that controls the size of the artificial interface boundary between SIMPIC and other solvers. As SIMPIC is being used as a performance proxy and is not a representative mini-app, the size of the mesh in the mini-app does not correspond to the size of the mesh used in the industry-led problem using the production applications. However, the size of the
interface that the mini-coupler is operating on must be the same as in the industry-led full scale simulation. As a result, the parameter is used to set the size of an ‘artificial’ mesh, built by padding out the smaller internal SIMPIC mesh at startup, from which the boundary interface can be generated from and passed to the coupler unit.

**MG-CFD**

If a user wishes to run a simulation where all MG-CFD inputs are the same size, then the input method remains the same as before; the input file can be passed using the ‘-i’ flag at run-time. However, for configurations which require specific input for each MG-CFD instance, we modified the MG-CFD input method to read from a file called ‘mg_files.input’ if the keyword ‘file’ is inserted after the ‘-i’ run-time flag.

Figure 5.4 presents the contents of a typical mg_files.input file. This file is passed to all MG-CFD units, in which each reads from the line which corresponds to the its unit number in the simulation. If an MG-CFD instance is the 2nd unit in the simulation for example, then it will read from the 2nd line and load the mesh described in input-mgcfd_8m.dat. A NULL is used in lines which correspond to SIMPIC instances, since these read from their own input file as described above. Note that this input is ignored, so this NULL text can be anything the user wishes as long as it contains a newline character.

### 5.1.3 Other changes

A few other modifications were made to enable the full-engine mini-simulation to run smoothly. Configuration settings for CPX, such as enabling MUM, the tree-based search, and changing coupling frequency were moved to a dedicated coupler configuration header.
file called ‘coupler_config.h’, which can be included in each of the solvers rather than being a central entity being passed through method parameters. In addition, the SIMPIC code was tweaked to fix several out-of-memory errors that were occurring during standalone testing.

## 5.2 Configuration of the full-engine simulation

### 5.2.1 Combustion test cases

In the full engine simulation we are introducing a combustion component, namely the RR Pressure Solver. As previously stated, our aim is to replicate the performance of the industry code using the SIMPIC mini-app by adjusting its run-time parameters. Therefore before testing a mini-simulation which includes SIMPIC, we must first evaluate the performance of the pressure solver in isolation. In our initial performance analysis of the pressure solver, we will examine two test cases: (1) a single sector swirl combustion case, with 28M cells and 7M particles, and (2) a triple sector swirl combustion case with 84M cells and 21M particles [Sidey et al., 2017].
### Table 5.1: The list of mesh sizes used in the mini-simulation.

<table>
<thead>
<tr>
<th>Instance #</th>
<th>Application</th>
<th>Mesh size (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MG-CFD</td>
<td>8</td>
</tr>
<tr>
<td>2-12</td>
<td>MG-CFD</td>
<td>24</td>
</tr>
<tr>
<td>13</td>
<td>MG-CFD</td>
<td>150</td>
</tr>
<tr>
<td>14</td>
<td>SIMPIC</td>
<td>380</td>
</tr>
<tr>
<td>15</td>
<td>MG-CFD</td>
<td>150</td>
</tr>
<tr>
<td>16</td>
<td>MG-CFD</td>
<td>300</td>
</tr>
</tbody>
</table>

#### 5.2.2 Setup and challenges of the full-engine simulation

Our motivating problem is a representative 1.5Bn cell test case from Rolls-Royce plc., consisting of 13 compressor rows, a combustor chamber, and two turbine rows. Figure 5.5 presents the layout of the simulation in both the industry simulation, which we call the full-scale simulation, and the equivalent mini-app simulation. The full-scale simulation consists of coupled instances of the RR Density Solver [Lapworth, 2004], with combustion modeled using the RR Pressure Solver [Anand et al., 2013], and coupling handled by the RR coupler framework [Amirante et al., 2021a], all of which we have discussed previously in Chapter 2. Work is currently in progress to run this test case with these industry applications. The mini-app simulation consists of 13 instances of MG-CFD, coupled to a SIMPIC instance, connected to 2 instances of MG-CFD which model the turbine. The coupling is handled by the CPX mini-coupler.

In comparison to the compressor test case modeled in Chapter 4, this is a geometrically smaller problem, but due to the inclusion of the compressor, combustor, and turbine, it is particularly challenging from a load-balancing and stability perspective. Although the specific dimensions of the motivating problem are not publicly available, Table 5.1 presents the size of each mini-app mesh. Each mesh is comparable to the industry test case, highlighting the significant variation in scale between application instances. The size difference between the Instance 1 and Instance 16 mesh is 38x; in contrast, the difference between the smallest and largest mesh of the full-annulus Rig250 compressor test case is only 2.7x.
Table 5.2: A comparison between the pressure solver test cases and the equivalent SIMPIC test cases (Base-STC) which replicate the performance behavior.

<table>
<thead>
<tr>
<th>Pressure solver mesh size</th>
<th># SIMPIC Cells</th>
<th>SIMPIC Particles per cell</th>
<th>SIMPIC Time-steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>28M</td>
<td>512,000</td>
<td>100</td>
<td>50,000</td>
</tr>
<tr>
<td>84M</td>
<td>512,000</td>
<td>300</td>
<td>50,000</td>
</tr>
<tr>
<td>380M</td>
<td>512,000</td>
<td>1800</td>
<td>50,000</td>
</tr>
</tbody>
</table>

As discussed before, a coupled simulation progresses by exchanging information at some regular frequency between the coupled applications through coupler units. The frequency of exchanges is determined by each application; for example the density solver, after each iteration would communicate with the pressure solver, awaits for the pressure solver to complete some number of iterations, and return results before progressing to the next iteration. This cascading dependency down a sequence of application instances results in the overall simulation progressing at the speed of the slowest component. This is challenging as not only do the mesh sizes differ greatly between instances, so too does frequency of the iterations; for each density solver time-step, there are 2 pressure solver time-steps. The size of the coupler unit interfaces also vary, as well as their frequency; instances of coupled density solvers must exchange data every iteration of the solver, whereas the coupling between pressure and density solvers only exchange every 20 iterations.

5.3 Building the performance proxy and identifying the simulation bottleneck

To turn SIMPIC into a generic PIC mini-app to a performance proxy, we will need to hand pick a set of parameters, seen in Figure 5.3, such that running SIMPIC with these parameters can predict the performance of the RR pressure solver. Thus, we must first profile the RR Pressure Solver code using the test cases described in Section 5.2.1. These were run for 10 timesteps, measuring the run-time excluding the pre- and post-processing.
Figure 5.6: Parallel efficiency of the pressure solver and SIMPIC on ARCHER2.

costs (which are fixed). As in previous tests, the hardware used was the 740k core HPE-Cray EX supercomputer, ARCHER2 [EPCC, Accessed May 2022], which uses nodes of 2×64C AMD EPYC 7742 2.25GHz microprocessors, with 256GB memory per node. All codes were compiled using CRAY 10.0.4 Fortran/C++ compilers and MPICH 8.0.16 MPI. As seen in Figure 5.6, parallel efficiency of the pressure solver drops below 50% at 3000 cores. Despite the increased size of the 84M test case compared to the 28M, the drop in performance is similar. This is significant as the full-scale test case for the pressure solver is large at ~400M cells (comparable to that of the CERFACS integrated model [Arroyo et al., 2021]), indicating parallel efficiency will drop off at a similar point.

To predict the 28M and 84M pressure solver run-time with SIMPIC, we hand picked the parameters seen in Table 5.2. Figure 5.6 and 5.7 shows the SIMPIC parallel efficiency and speedup compared to the pressure solver. SIMPIC is able to predict the run-time for the pressure solver across both test cases with a mean error of less than 9% and a worst case error of 22%. This error is a result of SIMPIC’s parallel efficiency drop-off being
marginally less steep than the pressure solver.

We can generate an indication of the pressure solver speedup in the full-scale 380M cell test case from the parameters in Table 5.2. Figure 5.8 shows the results of this test, from 1,000 to 10,000 cores (8 to 80 ARCHER2 nodes). Parallel efficiency approaches 50% at only 10,000 cores, suggesting a maximum speedup of about $6 \times$ in this test case. We will refer to this test case as the Base SIMPIC Test Case (Base-STC), as it represents the baseline pressure solver performance without any optimizations or improvements.

### 5.3.1 Simulation bottleneck

The industry simulation is made up of three core components: the Density Solver, the Pressure Solver, and the coupling framework. The existing research referenced in Chapter 4 demonstrated 88% parallel efficiency of the Rig250 Full Annulus simulation at $\sim$10,000 cores [Mudalige et al., 2022], and showed equivalent efficiency with smaller test cases. Thus, we can conclude that similar level of parallel efficiency can be achieved in the
compressor and turbine stages of the proposed full-engine test case in this work. We also saw in Chapter 4 that the coupling overhead was at most 15% with up to 64 coupler units at 512 nodes, and that was in a test case considerably bigger than the Density Solver meshes in the full-engine motivating problem. In comparison, the results both here in Figure 5.6 and in existing research suggests the scalability of the pressure solver application used in the combustor is comparatively worse; in an 84M cell test case, the parallel efficiency dropped below 50% parallel efficiency at only 896 cores [Thari et al., 2022]. Even when improvements were made to eliminate the overhead of the particle load balancing, parallel efficiency still dropped below 80% at the same 896 cores. We can therefore conclude that the pressure solver will be the bottleneck in a coupled HPCp-Combustion-HPT simulation.

**Figure 5.8:** The speed-up of SIMPIC with the representative large base test case on ARCHER2.
5.4 Further performance analysis and optimization of the Pressure Solver

The pressure solver models motion of fluid, as a result of the combustion of fuel, by solving the Navier-Stokes equations using pressure correction methods. It employs a LES approach, including a k-ε model for modeling turbulence, a common turbulence model used in industry. A Finite-Volume approach is used for the geometry. Fuel spray droplets are handled using a Lagrangian approach, where particles are moved every time-step once the other fields have finished updating, as noted in Figure 2.9. Combustion-turbulence interaction is implemented with a variety of standard CFD-Combustion models (Eddy break-up, Probability Distribution Functions (PDFs)) [Anand et al., 2013]. Modeling combustion and turbulence is computationally expensive, but assumptions can be made to simplify transport equations using scalar and velocity components [Versteeg and Malalasekera, 2007]. However, solving the pressure equation requires many iterations and remains an expensive operation [Shyy et al., 1992].

To understand the parallel efficiency behavior of the pressure solver, the code was profiled using ARM MAP [ARM, Accessed May 2022] using the small 28M mesh on 2048 cores or 16 ARCHER2 nodes. Above 2048 cores the parallel efficiency drops below 50%, as Figure 5.6 highlights, hence this configuration was specifically profiled.

Figure 5.9 presents the results from the profiling where we plot the run-time of each of the main functions in the production code relative to the total run-time. These are further split into compute and communication time. 46% of the run-time is spent updating the pressure field, with 21% of it being spent in MPI communications and 25% in compute. The pressure field routines use a Conjugate Gradient solver with Aggregate Algebraic Multigrid (AMG), thus it is likely that the bulk of communications time is being spent in near-neighbor data exchange in Sparse General Matrix-Matrix Multiplication (SpGEMM) and Sparse Matrix–Vector Multiplication (SpMV) operations. Further profiling has shown the bulk of compute time is spent iterating through multi-grid cycles and in the setup phase, including calculating the Galerkin coarse grid operator. The particle spray routine is the next most time-consuming routine, spending 96% of its run-time in communications. Other work profiling the pressure solver has shown that this is the
result of poor distribution of the particles across cores [Thari et al., 2022].

As the velocity fields and scalar calculations scale well, we do not focus our efforts on these methods when attempting to improve performance. Instead we focus exclusively on the particle component and the pressure fields. Some of the optimizations we will discuss can also be applied in other methods, specifically those that accelerate SpMV operations, however when estimating the performance of a theoretical optimized pressure solver, we will use the parallel efficiency of these other components from the base code.

5.4.1 Spray methods

Figure 5.10 gives a further breakdown, detailing the parallel efficiency of each of the methods from 128 to 2048 cores, as well as the overall parallel efficiency curve. The same 28M mesh is used. It is clear that the particle component, modelling the fuel spray, is the biggest bottleneck in the pressure solver application, dropping below 50% parallel efficiency at just 2 ARCHER2 nodes, or 256 cores.
Figure 5.10: A breakdown of the most time-consuming pressure solver functions, examining parallel efficiency of each function from 128 to 2048 ARCHE2 cores in a 28M cell test case.

High communication overhead is common with codes that contain a particle-in-cell component, due to challenges handling load balancing and information exchange with the solver efficiently. One approach, spacial partitioning, partitions the solver space, with each MPI rank handling the particles that belong in its partition [Pankajakshnan et al., 2011]. This approach, used by the pressure solver, struggles when particles are not evenly distributed across the partitions. An alternate approach, particle sharing, distributes the particles across ranks regardless of their location [Shigeto and Sakai, 2011]. However, this approach requires collective operations which can significantly degrade performance at high core counts. As a result, spatial partitioning is most commonly used, often with hybrid MPI+OpenMP to take advantage of shared memory space and improve cache reuse [Bettencourt et al., 2021].

Recently, an asynchronous task based approach was tested on the pressure solver application, dividing the MPI space into distinct spray and solver communicators [Thari
et al., 2021]. With this method, the two components run independently, synchronizing using one-sided MPI shared memory communicators introduced in MPI-3 [Gropp, 2012]. This approach was demonstrated to work at scale [Thari et al., 2022], hence we include it as one of the optimizations to apply and analyze in the pressure solver.

5.4.2 Pressure field

Figure 5.10 shows that the pressure field component also suffers a drop in parallel efficiency, albeit not to the same extent as the spray routines. Even with perfect scaling of the particles, the code will still drop to ~60% parallel efficiency at only 2048 ranks, as shown in Figure 5.11a. Thus, optimizing the spray alone will likely not be enough to ensure the pressure solver not being a bottleneck in the full-scale test case.

Due to the nature of the pressure equation being solved in the pressure field method, the solver requires many more iterations to converge than other calculations such as momentum or scalar transport equations [Shyy et al., 1992]. Thus, a multi-grid method must be used to improve the convergence rate. Multi-grid techniques work by gradually lowering the resolution of the grid, solving the set of equations, and then interpolating the grid back up to its original resolution [Thakur et al., 1996]. As the pressure solver being examined is unstructured [Anand et al., 2013], it uses AMG, which applies this technique to the system of equations themselves, rather than the geometry [Park et al., 2015].

Further profiling of the pressure field methods indicate the majority of time is spent in this process, thus we focus on accelerating the AMG, with an emphasis on accelerating SpGEMM and SpMV operations as these are also used in other methods such as those solving transport equations [Trias et al., 2022]. There is a vast body of research optimizing these routines [Park et al., 2015] [D’Ambra et al., 2021]. Here we focus on the most significant optimizations:

AMG setup

• For the smoother, Hybrid Gauss-Seidel should be used, employing Gauss-Seidel within a task but Jacobi methods across parallel tasks. This leads to better convergence within each multi-grid cycle provided the problem size is sufficiently large [Baker et al., 2011].
Within each MG cycle, the grid is restricted, solved, and interpolated until the finest mesh is created. We propose to use extended+i methods in the interpolation, which considers not only neighbors of a grid-point but also its neighbors’ neighbors. While this is more computationally expensive, it also accelerates convergence [De Sterck et al., 2008].

In some AMG solvers, the MG cycle (V-Cycle) convergence time can be reduced using Krylov subspace acceleration, known as a K-cycle. However, this method can result in poor scalability of the solver with large numbers of cores [D’Ambra et al., 2021]. We therefore recommend using V-Cycles with smoothed parallel pattern matching [D’Ambra and Vassilevski, 2013].

Matrix and vector multiplication

In traditional SpGEMM operations, the input matrices must be read twice: once to determine the size of the output matrix, and again during the multiplication [Patwary et al., 2015]. Instead, each thread can be allocated a large chunk of memory and the disjoint results can be copied to a contiguous memory space [Park et al., 2015].

To reduce the amount of branching in SpGEMM operations, a Sparse Accumulator (SPA) can be constructed which allows access of any matrix element in constant time [Park et al., 2015] [Gustavson, 1978].

During interpolation and restriction, which uses SpMV, values at the same points are mapped directly to the mesh above or below. As a result, the matrix can be rearranged such that the first rows are an identity matrix, which reduces computation and saves memory bandwidth [Park et al., 2015].

When AMG is used in a distributed system, matrix rows are spread across cores, in a compressed sparse row format. During a SpGEMM operation, a thread must renumber its column mapping array as it is likely it has received new values in the halo exchange. This is an expensive process as efficient parallel reordering is difficult to achieve. Instead, each thread can build a hash map, which is then merged into
a global array using a parallel merge sort. Using a reverse mapping, the values can be distributed back to the relevant threads [Park et al., 2015].

### 5.4.3 Extrapolating improvements

![Graph showing predicted parallel efficiency of optimizations](image)

**Figure 5.11:** Predicted pressure solver parallel efficiency before and after particle and solver optimizations.

When applying the particle optimizations to our previous results in Figure 5.10, we set the parallel efficiency of the spray routines to 100%, as the research indicates there is little difference in scaling between the pressure solver with the optimized spray and the same code without any spray routines [Thari et al., 2022]. For the pressure field, the AMG setup changes are required such that the matrix and vector multiplication optimizations work as effectively as possible, so these do not directly affect runtime. With matrix and scalar vector multiplication changes, the research [Park et al., 2015] indicates a speedup of $\sim3.5 \times$ over the base solvers at 128 nodes, limited by communication in halo exchanges. Since our cluster has $\sim9 \times$ the cores per node as in the research, we expect performance to be
more in line with the parts of the AMG solver which are less affected by communication, which give an average speedup of $5 \times$. We therefore apply a $5 \times$ speedup to the pressure field to approximate the benefits of the improvements with this optimization. The results from the the estimated performance gains of these optimizations are shown in Figure 5.11. While this is only an approximation, it is clear from the above related work/research that there is potential for significantly improved scalability compared to the base application.

A SIMPIC configuration can be generated to match the parallel efficiency of the above estimated optimizations for the pressure solver. This consists of 1.18M cells, 60,000 particles per cell, and running for 450 time-steps. The configuration predicts the estimated optimized pressure solver run-time with an error of less than 7%. The speedup and parallel efficiency of both can be seen in Figures 5.12 and 5.13. We will refer to this test case
as the Optimized SIMPIC Test Case (Optimized-STC), as it represents the potential performance of the industry Pressure Solver if a series of optimizations were applied.

5.5 Extending the empirical performance model

Having created two SIMPIC test cases, one for modeling the original Pressure Solver (Base-STC) and the other for modeling a potential optimized Pressure Solver (Optimized-STC), we next need to add the ability to predict their run-time and performance characteristics to empirical performance model. The process we will follow will be the same as Section 3.2; we will first generate an initial run-time for the test cases, and pick parameters for the equation to model the parallel efficiency.
5.5.1 Initial run-time

Although the model is predicting the mini-simulation run-time and performance, the aim is ultimately for the model to load balance and predict the run-time of the full scale simulation with the industry applications. Thus, we must first convert the Density Solver time-steps and iterations parameters in the model ($N_{TS}$ and $I_{TS}$) to Pressure Solver time-steps, as the Pressure Solver time-steps take half as long and therefore 2 must be ran for every Density Solver time-step. From there, we can scale the initial Base-STC and Optimized-STC run-times, which are representative of a set number of Pressure Solver time-steps.

For the Base-STC initial run-time, $RT_{Base-STC}$, we used the test 28M test case parameter set from Table 5.2, ran on 1 ARCHER2 node. This was equivalent to 10 Pressure Solver time-steps, and produced $RT_{Base-STC} = 800s$. For the Optimized-STC initial run-time, we used a 28M test case as the base size. As we only have an Optimized-STC parameter set for a large instance, we applied the single node performance improvements found in the research from Section 5.4 of $\sim$40% to the $RT_{Base-STC}$ time to get $RT_{Optimized-STC} = 480s$. Initial testing found that at large mesh sizes the model was overestimating run-time; for example, at 380M cells, the model’s prediction was approximately $2.5 \times$ the actual result. Thus, the prediction is multiplied by an additional adjustment factor when the mesh is larger than 84M cells. This adjustment factor can be computed by:

$$A_f = \frac{1}{(\log_e \left( \frac{S_{mesh}}{28} \right))}$$

The initial coupling run-time is now changed depending on whether a sliding plane or overset-style steady state approach is used, as well as accounting for the tree-based search mentioned in Section 4.4.2. Compared to the original model example in Figure 3.6, the search time is now multiplied by $n\log n$ rather than taken to a power, and is only multiplied by $N_{TS}$ if the user specifies sliding plane, otherwise the multiplicative factor is removed.
5.5.2 Modeling SIMPIC parallel efficiency

Table 5.3: The parallel efficiency, or ‘scaling factor’ equations for the Base-STC and Optimized-STC

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol+Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized-STC P.E</td>
<td>$SF_{main} = 1 - 0.01 \times \left( \frac{R_{main} - 127}{214} \right) / \log_e (Scale \times 28 + e) \right) \right) \right) \right) \right) \right)</td>
</tr>
<tr>
<td>Base-STC P.E ($\leq$84M)</td>
<td>$SF_{main} = 1 - 0.01 \times \left( \frac{R_{main} - 127}{45} \right) / \log_e (Scale \times 28 + e) \right) \right) \right) \right) \right) \right)</td>
</tr>
<tr>
<td>Base-STC P.E (&gt;84M)</td>
<td>$SF_{main} = 1 - 0.01 \times \left( \frac{R_{main} - 127}{90} \right) / \log_e (Scale \times 28 + e) \right) \right) \right) \right) \right) \right)</td>
</tr>
</tbody>
</table>

Table 5.3 presents the parallel efficiency equations for SIMPIC in the model. The first equation is for modeling the parallel efficiency of the Optimized-STC, with the second and third for modeling the Base-STC for small and large test cases. In the model there is no transition between the two Base-STC equations, since we are not using an intermediary test case. However if this were the case, such as if a 90M node input was tested, a weighting function could be used to create a smoother transition. One change from the MG-CFD equation is that the initial run-time was executed using 1 ARCHER2 node, of 128 cores, rather than 100 ARCHER2 cores. This is because much of the testing in the initial MG-CFD and CPX work was done before ARCHER2 came online, so 100 cores was chosen as the initial case.

5.5.3 Other changes

The run-time derivation from Section 3.2.3 remains unchanged. The parallel efficiency is plugged into an equation which uses the original run-time and number of cores to calculate the new run-time given a certain core allocation. However, since variance in mesh size is now large, as discussed in Section 5.2.2, the model now supports a different mesh and interface size for each solver and coupler instance. The result is that each solver and coupler unit have their own initial run-time and parallel efficiency equation, rather than just two global equations for all of the solvers and all of the coupler units, as was the case previously.
5.6 Applying the extended performance model

5.6.1 Building the iterative algorithm

**Algorithm 4:** Distribute ranks to coupled mini-apps

**Input:** $T[\text{apps}]$, $\text{Size}[\text{apps}]$, $\text{Size}[\text{CUs}]$, $\text{Iter}[\text{apps, CUs}]$

**Result:** Optimized core distribution and predicted run-time of coupled simulation

for $i = 1$, Num. of mini-app instances do

$T[\text{app}_i] = T[\text{app}_i] \times \frac{\text{Size}[\text{app}_i]}{\text{Base\_size}} \times \frac{\text{Iter}[\text{app}_i]}{\text{Base\_iter}}$

end

for $i = 1$, Num. of coupler units do

$T[\text{CU}_i] = T[\text{CU}_i] \times \frac{\text{Size}[\text{CU}_i]}{\text{Base\_size}} \times \frac{\text{Iter}[\text{CU}_i]}{\text{Base\_iter}}$

end

$ranks \leftarrow \#\text{ cores}$;

while $ranks > 0$

$\text{App}_{\text{max}} \leftarrow \text{MAX}(T[\text{app}_1], T[\text{app}_2], ..., T[\text{app}_n])$;

$\text{CU}_{\text{max}} \leftarrow \text{MAX}(T[\text{CU}_1], T[\text{CU}_2], ..., T[\text{CU}_n])$;

$\text{App}_{\text{diff}} \leftarrow T(\text{cores}[\text{App}_{\text{max}}]) - T(\text{cores}[\text{App}_{\text{max}}] + 1)$;

$\text{CU}_{\text{diff}} \leftarrow T(\text{cores}[\text{CU}_{\text{max}}]) - T(\text{cores}[\text{CU}_{\text{max}}] + 1)$;

if $\text{CU}_{\text{diff}} > \text{App}_{\text{diff}}$ then

$\text{cores}[\text{CU}_{\text{max}}] \leftarrow \text{cores}[\text{CU}_{\text{max}}] + 1$

else

$\text{cores}[\text{App}_{\text{max}}] \leftarrow \text{cores}[\text{App}_{\text{max}}] + 1$

end

$ranks = ranks - 1$

end

$\text{output} \leftarrow \text{MAX}(\text{app}_1, ..., \text{app}_n) + \text{MAX}(\text{CU}_1, ..., \text{CU}_n)$;

$\text{output} \leftarrow \text{cores}[\text{apps}], \text{cores}[\text{CUs}]$

Algorithm 3, which distributed resources for the prior compressor problem, allocated a core to either all the coupler units or all MG-CFD instances every iteration; the choice was dependent on whichever allocation resulted in the largest drop in run-time. As the run-time of that coupled simulation was the sum of coupler and MG-CFD time, this resulted in the lowest overall run-time. However, when incorporating different solvers, mesh sizes, coupling types and interface sizes, this approach is no longer viable. Instead, the algorithm must choose the slowest solver instance, determine the reduction in run-time
if a core is allocated, and then compare against the reduction in run-time in the slowest coupler. The total run-time is the sum of the slowest solver and the slowest coupler unit, so the comparison must be made between these two entities in order to result in the lowest overall run-time.

Algorithm 4 implements the extended model. The initial run-times for each of the solvers and coupler units, as well as the size of the mesh input and coupler interfaces are calculated and stored in arrays. For each iteration, the algorithm compares the run-times of each mini-app instance, selecting the instance with the longest run-time as well as the slowest coupler unit. The program then compares the reduction in run-time of allocating a core to both, choosing the option where the reduction in run-time is the greatest. As the run-time of the simulation is the sum of the slowest mini-app instance and the slowest coupler unit, this results in the greatest reduction of run-time for every core allocated.

Figure 5.14 shows a typical iteration in the extended resource allocation algorithm. In this simulation example, two MG-CFD units are connected to a SIMPIC unit, with the left MG-CFD unit operating on a domain size of 8M nodes and the right MG-CFD unit operating on a domain size of 28M nodes. The SIMPIC domain size is 28M nodes. The MPI ranks of each solver and coupler unit are displayed above the simulation diagram, with the corresponding run-time displayed below. At the start of the iteration, the model compares the individual run-time each of the solver instances and selects the slowest. This step repeated with the coupler units. In Figure 5.14, the first MG-CFD unit and second coupler unit (highlighted in green) are the slowest component of their respective groups. The model then allocates a rank to each and compares the reduction in run-time over the original, shown in step 3 of the diagram. In this case, the allocation to the MG-CFD unit delivers a reduction in run-time of 1 second, and the allocation to the coupler unit reduces its run-time by 4 seconds. Thus, the allocation to the coupler unit delivers a greater reduction in run-time, so this is the unit where the rank is to be allocated. In the final step, the model has allocated this rank and the run-time is updated.
5.6.2 Iteration example

Figure 5.14: An example of a loop iteration of the extended resource allocation algorithm.
5.6.3 Comparison to original algorithm

These changes present several significant improvements over Algorithm 3. In the original implementation, all instances of the solvers must have the same mesh and interface size. We have extended the model so each mini-app instance can have a different mesh size and interface size. As a result, run-time comparisons and core allocation is done on a per-instance basis, rather than crudely allocating resources to either all solver instances or all coupler instances. Additionally, the model now supports instances of both pressure-solver and density solver mini-apps, whereas before it could only generate resource allocation and run-time predictions of coupled density solver mini-simulations. The extended model, therefore enables rapid resource distribution decision making for a full HPCp-Combustor-HPT simulations, facilitating load-balancing for more complete workloads and real-world scenarios.

5.7 Evaluating the full engine test cases

So far, we have examined how the CPX mini-coupler can be extended to build a mini-simulation of a full gas turbine using existing representative mini-apps and a performance proxy mini-app SIMPIC. The design of the upgraded mini-coupler was detailed, including how the SIMPIC application was integrated, and how the framework handled input from many different solvers. The industry-led problem and equivalent mini-app test case was detailed, and the parameter set required for SIMPIC to be used as a performance proxy was produced. The bottleneck of the simulation, the Pressure Solver, was identified and profiled, and a number of existing optimizations were examined. By extrapolating the results in wider research and applying them to our results, we speculated on the predicted speedup that an ‘optimized’ pressure solver would achieve and produced a SIMPIC test case which could predict this run-time. In addition, the empirical performance model detailed in previous chapters was extended to incorporate the new solver type and the improved tree-based search. An upgraded iterative algorithm which implements the model was also presented, with the ability to generate a run-time prediction and a core distribution in a simulation with different mesh sizes, interface sizes and coupling frequencies.

We have previously demonstrated the accurate simulation and load balancing of a
large-scale gas turbine compressor using mini-apps in Chapter 4. However, as discussed in
Section 5.2.2, achieving this with a full-engine simulation is significantly more challenging.
Thus, we seek to evaluate the resource allocation and performance prediction capabilities
of the extended mini-simulation and the empirical performance model detailed thus far.
Although there are no large scale coupled industry results to directly compare against,
as this work is ongoing, we believe that using the existing results of Chapter 4 along
with comparisons between the mini-apps and the performance model will provide valuable
insight as to the behavior of the simulation when using the original and optimized pressure
solver. Section 5.8 compares the core allocation, predicted run-time and run-time of the
mini-apps in a mini-simulation using a small 3-stage test case. Section 5.9 repeats this
process but for the full 16-stage full-engine test case as discussed in Section 5.2.2. Finally,
Section 5.10 summarizes the results.

5.8 Comparison with a small 3-stage test case

Table 5.4: The mesh size and MPI ranks of each component in the small
3-stage mini-app simulation.

<table>
<thead>
<tr>
<th>Instance #</th>
<th>Application</th>
<th>Mesh size (millions)</th>
<th>MPI Ranks (Base-STC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPIC</td>
<td>28</td>
<td>4,253</td>
</tr>
<tr>
<td>2,3</td>
<td>MG-CFD</td>
<td>150</td>
<td>311</td>
</tr>
</tbody>
</table>
Figure 5.15 compares a run-time prediction in the extended predictive model with the mini-app runs in a typical medium-size High-Pressure Turbine test case. This scenario consists of 1 SIMPIC unit and 2 MG-CFD units, with 5,000 cores allocated to the total simulation. The MG-CFD meshes are instances solving the NASA Rotor37 [Denton, 1997] meshes (150M nodes), which represent the geometry of a transonic axial compressor rotor, widely used for validation in CFD. The SIMPIC test case represents a pressure solver over 28M cells. 331 ranks are allocated per MG-CFD unit, with 4,253 allocated to SIMPIC, shown in Table 5.4 For this validation, SIMPIC is configured in its Base-STC setup. 63 CU are used between the MG-CFD units, with 22 CU being allocated between the SIMPIC and MG-CFD units. As can be seen from Figure 5.15, the performance model is able to load balance the applications and predict the run-time with a maximum error of 18%.

These initial tests highlight the limitations of the existing pressure solver, with 85% of the cores being allocated to SIMPIC, despite the mesh size being less than \( \frac{1}{5} \)th the size of the MG-CFD domain. This suggests that similar to the brute force coupler unit search in Chapter 4, it is impractical to run a large coupled simulation using the unop-
timized Pressure Solver due to the resources required. Our next task is to examine the resource allocation requirements and generate a performance prediction of one of these large coupled simulations.

5.9 Comparison with the 16-stage full-engine test case

The second validation case attempts to replicate the HPCp-Combustor-HPT setup from Figure 2.10. The mesh sizes for each component simulation is detailed in Table 5.5. Note that the listed SIMPIC mesh size is the equivalent cell size in the full application; the actual input size that represent performance of a 1.2M cells problem. However, we choose to quote 380M since the size of the interfaces passed to the coupler units are made to match the full-scale simulation. In total, the effective size of this mini-app simulation is 1.25Bn cells, equivalent to our main motivating problem from industry.

Research suggests that beyond $\sim 30,000$ cores, the setup cost of the AMG in the pressure solver becomes significant [D’Ambra et al., 2021]. As a result, we enter these parameters into the model with a core allocation budget of 40,000 cores; $\sim 30,000$ cores for SIMPIC and $\sim 10,000$ cores to the rest of the simulation. The simulation was ran for the equivalent of 20 time-steps of the pressure solver, comparing the model’s predicted run-time of the simulation with the standalone run-time of each mini-app instance. The number of cores allocated to each instance by the model is shown in Table 5.5, done such that in the full, longer mini-app simulation, we have confidence that the resource allocation algorithm will work correctly. Figure 5.16 shows percentage error of each instance size for mini-app simulations using the Base-STC and the Optimized-STC. It demonstrates the model can predict the run-time of all instances used in both mini-apps and by extension full-scale simulations with a 25% worst case error and a 12% mean error.
Figure 5.16: The percentage error between the mini-apps and predictive model for individual mini-app simulations using the Base-STC and the Optimized-STC.

Table 5.5: The mesh size and rank allocation of each component in the full 16-stage mini-app simulation

<table>
<thead>
<tr>
<th>Instance #</th>
<th>Application</th>
<th>Mesh size (millions)</th>
<th>MPI Ranks (Base-STC)</th>
<th>MPI Ranks (Optimized-STC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MG-CFD</td>
<td>8</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2-12</td>
<td>MG-CFD</td>
<td>25</td>
<td>100</td>
<td>163</td>
</tr>
<tr>
<td>13</td>
<td>MG-CFD</td>
<td>150</td>
<td>167</td>
<td>1,218</td>
</tr>
<tr>
<td>14</td>
<td>SIMPIC</td>
<td>380</td>
<td>13,428</td>
<td>32,201</td>
</tr>
<tr>
<td>15</td>
<td>MG-CFD</td>
<td>150</td>
<td>167</td>
<td>1,218</td>
</tr>
<tr>
<td>16</td>
<td>MG-CFD</td>
<td>300</td>
<td>338</td>
<td>3,357</td>
</tr>
</tbody>
</table>

The same setup was then ran for the equivalent of 1,000 time-steps of the density solver,
which is the time taken for half a revolution of a gas-turbine engine. The length was chosen as it is large enough to give representative data without wasting HPC resources. With the large mini-app simulation using the Base-STC example, the bottleneck was SIMPIC, which reached 50% parallel efficiency at 13,428 ranks, shown in Table 5.5. The maximum error between the overall predicted run-time and the run-time of the full proxy simulation was less than 25%, shown in Figure 5.16.

The model starts rank allocation at 100 for this problem size as mini-app base cases run in a sufficiently short time, yet have a higher parallel efficiency at 100 ranks. As a consequence, the MG-CFD 8M instance gets allocated more ranks than necessary to match the run-time of the overall simulation. However, in this base test case, the only place to re-allocate additional ranks to improve run-time would be to SIMPIC, and as the parallel efficiency of the SIMPIC instance is already at 50%, the impact on overall run-time would be negligible. Thus, in the full-scale simulation, we predict the pressure solver application would be the bottleneck in the simulation.

In the mini-app simulation using the Optimized-STC, the model predicts that both

\[ \text{Figure 5.17: The predicted and actual speedups for 1 revolution of the mini-app simulation using the Optimized-STC compared to a mini-app simulation using the Base-STC.} \]
the density and pressure solvers will scale well at 40,000 cores. The model predicts that with 32,201 cores allocated, or \(\sim 262\) ARCHER2 nodes, parallel efficiency of the pressure solver will be 87\%, this being the worst out of all the application instances.

The model predicts coupling overhead to be \(<0.5\%\) of overall run-time. This is in contrast to the significant bottleneck predicted by the model in [Powell et al., 2021]. We attribute the reduction in overhead due to the improved search algorithm, with a tree-based search routine and by pre-fetching the cells required for the next iteration, for sliding-planes interaction which was subsequently implemented into the industrial coupler [Mudalige et al., 2022].

Comparing the two test cases, Figure 5.17 presents the predicted speedup in a 1 revolution simulation if an optimized pressure solver was used in place the original. The model predicts that by optimizing the pressure solver with the improvements discussed in Section 5.4, the simulation can be sped up significantly by a factor of \(6\times\).

Figure 5.17 compares the predicted speedup with the measured speedup on ARCHER2 for both the Base-STC and Optimized-STC mini-app setups. To save time on the HPC system, the measured run-time is taken from a 0.5 revolution run and then doubled to match the 1 revolution prediction. The model is able to predict the run-time of both scenarios with a maximum error of less than 25\%, and demonstrates a speed-up of approximately \(~4\times\) when comparing the Optimized-STC simulation to the coupled mini-app simulation using the Base-STC. The difference between the predicted and actual mini-app simulation run-time appears mostly a result of SIMPIC’s base test case running faster than expected. Input parameters required to match the base pressure solver scaling, which are not typical of a real world electrostatic problem, cause run-time per time-step to improve when the number of time-steps are significantly increased. This behavior is not seen in the production pressure solver, and highlights a limitation with using a mini-app that was not derived from the production application. Despite this, the predictive model and large mini-app run suggest a possible speedup of between \(4\times-6\times\).

It is important to be aware of the other limitations with the ‘performance proxy’ approach presented here. The optimizations presented may deliver different parallel efficiency improvements, and the performance characteristics may not resemble those that were predicted. In the ideal case, where the speedup matches the best cases from the quoted research, we predict overall speedup to be \(~7.5\times\) over the base case. In a worst
case scenario, where the particle optimizations are applied, pressure field run-time is reduced only by 30% and pressure field parallel efficiency doesn’t improve from the base case, we predict speedup to be only 2.3×. The results supporting these optimization improvements are available only in the mini-app domain, as work running the full-scale simulation is still in progress. However, it is clear from profiling the pressure solver and existing research on the application itself that there is clear scope for efficiency gains.

5.10 Summary

In this chapter, we have presented the design overview and extensions required to build a full-engine mini-simulation using mini-apps and an extended empirical performance model. Following this, we have demonstrated the use of these mini-apps, the CPX mini-coupler, and the performance model to model and predict the run-time for a gas-turbine engine simulation with a high degree of accuracy. One significant finding was that using the original Pressure Solver in a coupled simulation of this scale was simply not practical, as could be seen by the results in Chapter 5.8 and reinforced by the resource allocation in Table 5.5. Despite the limitations of the performance proxy approach, it is clear that significant progress can be made in achieving high performance large scale coupled HPCp-Combustor-HPT simulations by optimizing the Pressure Solver code.
Chapter 6

Conclusions and Future Work

As computing power and our desire to model complex physical phenomena grows, concurrent multi-physics simulations are playing an increasingly important role in scientific computing. In this area, the challenge of performance prediction, load balancing and optimizing these simulations by hand is becoming intractable at current problem sizes and beyond. This is especially pertinent in the area of gas-turbine aero engines due to the complexities of the simulation, stemming from domains moving relative to one another, the number of solver instances and the intensity of coupling. In addition, the continuously changing landscape in HPC further adds to these challenges; the expanding number and variety of hardware platforms and parallel programming frameworks makes it difficult to achieve both high performance and performance portability. To get to extreme scales required for virtual certification and to continue to model physical phenomena with greater fidelity, and this is simply not possible using existing methods. The motivation of this research was to present tools and frameworks to help address these issues in order to enable these extreme scale simulations.

Importantly, the ideas presented are designed to be application independent such that they can be applied in a variety of current and future concurrent multi-physics simulations outside the scope of gas-turbine aero engine modeling. In addition, because of the variety of hardware and software platforms, this work is also designed to be platform and software independent, so that the techniques can be applied on a different hardware architecture or parallel programming model.
6.1 Contributions

To address the above concerns, the first part of the thesis focused on using mini-apps to build a proxy mini-simulation, which was representative of the full-scale simulation with production applications. Resource allocation would be handled by a predictive empirical model, using the mini-apps as reference points, which could allocate the cores to ensure optimal run-time. By doing so, the performance of the system and the components could be evaluated without using production applications or production settings.

The first contribution of this work is the development of CPX, a representative coupler mini-app designed to match the performance behavior of Rolls-Royce’s production coupler framework. As the framework is a mini-app, simplifications can be made to the setup of the coupling, enabling theoretical configurations which are not possible using the production tools to be explored. In particular, the ability to automatically partition coupling domains means resource allocation can be tested quickly and with a wide range of values. Although CPX is built to replicate the design philosophy and scalability of the RR coupler, its ideas can be applied in a wide range of coupled simulations outside this domain. The code is open source and available on GitHub to provide a starting point for such developments. CPX was combined with the MG-CFD mini-app to create proxy configurations to capture the operation of production compressor simulations, and the run-time and scaling behavior of MG-CFD with CPX was shown to have the same quantitative and qualitative behavior as Rolls-Royce’s production applications. An analytical comparison was used to demonstrate that for a certain allocation of coupling resources, CPX and RR Coupler will have similar scalability behavior. CPX and MG-CFD were then tested using a variety of configurations, and compared to the RR Coupler coupling the RR Density Solver, the mini-apps were able predict the run-time with a less than 17% mean error.

Along with the mini-coupler and mini-simulation, a further contribution is the specification and development of a general-purpose empirical performance model, designed to predict and allocate resources to the components of the coupled compressor simulation of the production applications using an iterative algorithm. These tools were developed using the insights from CPX and MG-CFD using a curve fitting approach, and the algorithm was implemented in Python. Like the mini-coupler, the design of the performance
model and iterative algorithm is application independent; the ideas can be applied to any parallel multi-physics environment. The model was used to predict optimum configuration settings on a cluster, and the prediction was then tested against other configurations on the 740,000 core ARCHER2 supercomputer [EPCC, Accessed May 2022] to show that the predicted configuration was optimal. The model was then used to predict the optimum configuration for a 6Bn cell test case using 100,000 cores, as well as the same run on a 400 GPU system.

Running these tests provided some insights about the current state of the full-scale compressor simulation; due to limitations with the number of coupler units between solver instances, it would be impossible to run this simulation at scale without design changes, specifically the brute-force search methods. If the proxy tools were not used, it would be extremely time consuming and likely not possible to determine this to be the case with production applications and settings. Comparing the run-time and scaling in an updated compressor simulation with tree-based search to an updated performance prediction further validated the accuracy of the proxy tools.

In the latter half of this work, the focus was on the significantly more complex multi-physics problem of a full gas-turbine engine simulation, motivated by industry challenges. The combination of different solver and coupling types and a large variation in size (>30×) between mesh instances makes load balancing and performance analysis extremely challenging. To enable this scale of simulation to be run, a number of changes were made to the tools. The mini-coupler was extended to include support of the SIMPIC mini-app, used as a performance proxy for the industry Pressure Solver code, as well as supporting different sized input for each solver and coupler instance. Alongside this, the empirical performance model was extended, adding the ability to predict SIMPIC (and by extension the RR Pressure Solver) performance and different input sizes for each input. The iterative algorithm was also modified to support these extensions.

Using these tools, we sought to understand and address the bottleneck in the system and speculate about performance improvements. After identifying the Pressure Solver to be the likely bottleneck in the system, we investigated optimizations in wider research, and extrapolated theoretical performance gains to the existing Pressure Solver code. Two SIMPIC test cases were created, one which was shown to predict the original pressure solver run-time with a maximum error of 22%, and another to match the performance
of a high performance pressure solver code. The predictive model was tested using a small coupled test case, predicting mini-app run-time with a maximum error of 18%. One interesting insight was that almost all of the compute resources needed to be allocated to the original pressure solver in order to obtain level performance with the density solver. A large coupled 16-stage full stage full-engine mini-simulation was executed on up to 40,000 cores, with the performance model able to predict the run-time of each individual component with max 25% error. This test highlighted that significant performance gains (4-6×) can be achieved and the bottleneck can be likely addressed with the proposed optimized pressure solver.

It is important to note that with representative mini-apps, the entire process of mini-simulation design, performance model building, profiling, optimization and resource allocation can be done using entirely open source and publicly available codes and input files. Although in this case we used the industry pressure solver for profiling, this is only because we did not have a representative mini-app for the pressure solver and relied on a ‘performance proxy’ approach with SIMPIC. While benchmarking of production tools is required during the design phase of a mini-app, the execution stage is independent, which can then be used to give insight on optimizations and resource allocation of a full-scale simulation using production applications. The ideas and methods presented in this thesis provide a blueprint on how to create the required mini-coupler framework and performance modeling tools, allowing the creation of optimal and representative mini-simulations. These mini-simulations are vital to the future of extreme scale multi-physics computing, as without them it is expensive, impractical and in some cases impossible to evaluate and optimize the performance of coupled systems.

6.2 Future Work

Until recently, the scale and complexities of multi-physics simulations were limited due to the computational requirements to achieve accurate modeling of physical phenomena. However, as the power of computing hardware grows, so too does the need for performance insight and effective resource allocation. We believe the techniques presented here provide a solid foundation to address these requirements, however there is considerable scope for improvements across each part of the presented ideas. We will focus on several areas where
there is strong motive for continued development. The first area relates to improving the
accuracy of the coupling and incorporating new coupling techniques to CPX. Additionally,
we will briefly examine methods to raise the accuracy of the performance model. Finally,
methods to further validate the existing techniques, and extensions to current gas-turbine
simulations will be considered.

6.2.1 Improving CPX and exploring different coupling methods

Our focus on the CPX mini-coupler throughout this thesis has been on capturing perfor-
mance trends of the industry coupler code and evaluating different resource allocations
without manually partitioning the domains. As a result, CPX was designed using a
‘top-down’ approach where the aim was to mimic the scaling and performance of the
search, interpolation and communication routines, rather than a ‘bottom-up’ approach
where the routines are mathematically accurate and the internals are very closely aligned
to the industry code. This ‘bottom-up’ approach brings the benefit of increased accu-
racy, at the expense of lack of flexibility such as having to partition domains manually,
as well as increased development time. At this stage, the most challenging issue fac-
ing many coupled multi-physics simulations is numerical stability [Arroyo et al., 2021],
so having a mathematically accurate mini-coupler could be beneficial despite its down-
sides. The Rolls-Royce industry coupling framework is a relatively small code compared
to the physics solvers, so the question remains whether a mini-coupler designed with this
approach is even necessary, and what simplifications could be made.

Both the RR coupler and CPX use an almost identical pure MPI approach, where the
communicator is split and data is sent explicitly between solvers and coupler units. As the
size of domains and the fidelity of meshes increase, the communication between solvers,
particularly in collective operations, may present a bottleneck in future simulations. In-
corporating a shared memory approach may help address this issue. Traditionally, shared
memory coupling was only possible with frameworks such as OpenMP [Theler et al.,
2013], which limits scalability beyond intra-node communication. This can be some-
what mitigated by using a hybrid MPI+OpenMP approach, although this still relies on
collectives and is often only to the benefit of memory-bound applications. However, in
recent years advancements have been made to support shared memory models outside

120
of these frameworks. MPI 3.0, released in 2012, added support for shared memory with
MPI communicators [Hoefler et al., 2013] [Dinan et al., 2016], which can bring signif-
icant performance gain when compared to point-to-point communication [Thari et al.,
2021]. Another technique is to use the job scheduler itself to split up allocate resources
to each solver, rather than doing this through MPI, removing any synchronization points
in the simulation [Slurm, Accessed July 2023]. These techniques could be integrated and
evaluated using the mini-coupler and relevant mini-apps.

6.2.2 Improving the accuracy of the performance model

A curve-fitting approach was used for the predictive performance model developed in
this thesis; the approach was chosen due to relatively high accuracy given the short
time duration required to choose the parameters. It is also general purpose, so any new
application or solver technique can be added to the model with relative ease. There are
two main shortfalls with this approach; the first is that as it does not take to consideration
of the internals or communication of an application, so accuracy will always be reduced
compared to more fine-grained techniques. In addition, the parameters for the curve-
fitting equations have to be chosen by hand, which is currently done by trial-and-error.

A different approach worth exploring would be to use analytical models [Clement and
Quinn, 1993], which parameterize not only the compute behavior but also communication
and network topology, memory and cache, and compiler optimizations. Although these
would require a substantial amount of work to design and implement, they would deliver
very high accuracy which may be required as the complexities of coupled multi-physics
simulations grow. The parameter exploration of such techniques can be accelerated by
using profiling tools such as Linaro Forge (formerly ARM Forge) [Linaro, Accessed July
2023] or Scalasca [Scalasca, Accessed July 2023] and feeding logs into the model. Addition-
ally, parameter selection in both a curve fitting or analytical approach can be accelerated
using AI heuristics and machine learning techniques to further speed up development
time [Memeti and Pllana, 2021].
6.2.3 Further validation and extended simulations

As work running a full-scale simulation with the internal Rolls-Royce applications is ongoing, our results presented in Section 5 were limited to comparisons between the performance model and the mini-simulation. While the results from the mini-coupler and MG-CFD were validated using the compressor problem in Section 4, validation with the entire engine is not currently viable. However, as development progresses and a run of the full-scale industry simulation becomes possible, further validation of the results from the model and mini-simulation can be performed. One limitation of the work is the use of the SIMPIC ‘performance proxy’, which compared to a representative combustion mini-app will be less accurate. Currently under development by the University of Bristol is a combustion mini-app that aims to model a high-performance combustion and particle code [UoB, Accessed September 2022], which can be substituted for SIMPIC as development progresses.

In addition, there is scope to extend the simulation by evaluating structural and thermal modeling of the engine during a CFD-combustion simulation using Finite Element Modeling. Current work is ongoing adding the open source Finite Element FEniCS-X [FEniCS, Accessed September 2022] code to the full-scale industry engine simulation, along side work integrating the code to the mini-simulation.

6.3 Final thoughts

Advancements in hardware and software platforms in High Performance Computing are enabling high fidelity multi-physics simulations which allow us to broaden our understanding of physical phenomenon. Such exciting developments unlock the potential for important advancements in science and engineering; these include significantly speeding up development time of physical prototypes, accurately forecasting weather events given changes in atmospheric conditions, and validating novel methods of power generation. However, the complexities of such simulations brings challenges with efficient execution, load balancing and bottleneck identification. The coupled mini-apps and performance model combination presented in this thesis demonstrates how rapid design space and run-time setup exploration studies can be carried out to obtain the best performance
from full-scale CFD-CFD and Combustion-CFD coupled simulations, and how to identify and address bottlenecks within the simulation. We believe that similar techniques can be applied to help solve other such multi-physics challenges, and look forward to the advancements in these areas.
Bibliography


EPCC. Archer2, Accessed May 2022. [https://www.archer2.ac.uk](https://www.archer2.ac.uk).


