UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND THE ENVIRONMENT

Fluid Structure Interactions Group

Chaotic Methods for the Strong Scalability of CFD

by

James Hawkes

Thesis for the degree of Doctor of Philosophy

August 2017

"Let's think the unthinkable, let's do the undoable. Let us prepare to grapple with the ineffable itself, and see if we may not eff it after all."

– Douglas Adams

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND THE ENVIRONMENT
Fluid Structure Interactions Group

Doctor of Philosophy

CHAOTIC METHODS FOR THE STRONG SCALABILITY OF CFD

by James Hawkes

Supercomputing power has been doubling approximately every 14 months for at least three decades, increasing the capabilities of scientific modelling at a similar rate. The first machines capable of one ExaFLOP (10^{18} floating-point operations per second) are expected by 2020. However, architectural changes required to reach 'exascale' are significant, with energy efficiency constraints leading to a huge growth in parallelization. A new era of computing has arrived, dubbed the 'many-core' era, in which the number of computing cores is increasing faster than CFD simulation sizes – prompting the research question for this thesis:

> *'What limits the strong scalability of CFD and its ability to handle many-core architectures? What can be done to improve the CFD algorithms in this respect?'*

A number of scalability investigations have been performed from 1 through to 2048 cores, using a semi-implicit, finite-volume CFD code: *ReFRESCO*; and the University of Southampton supercomputer: *Iridis4*. The main bottleneck to strong scalability is shown to be the linear equation-system solvers, occupying up to 95% of total wall-time on 2048 cores – where the poor scalability arises from synchronous, global, inter-process communications. Experiments have been performed with alternative, state-of-the-art linear solvers and preconditioners, without significant improvements, which motivates novel research into scalable linear solvers for CFD.

The theory of 'chaotic relaxation' has been used to create a completely asynchronous Jacobi-like 'chaotic solver', showing almost perfect scalability, and performance far greater than their synchronous counterparts. However, these solvers lack the absolute numerical power to compete with existing solvers, especially as the resolution of the simulations increases. Following this, chaotic relaxation theory has been used to create a novel 'chaotic-cycle' multigrid solver, combining aspects of the chaotic solver and classical multigrid methods.

Both of the solvers have been verified and tested using canonical test cases and practical CFD simulations. On 2048 cores, the chaotic-cycle multigrid solver performs up to 7.7× faster than a typical Krylov Subspace solver and 13.3× faster than classical V-cycle multigrid. With improvements to the implementation of coarse-grid communications and desynchronized residual computations, it is likely that the chaotic-cycle multigrid method will continue scaling to many thousands of cores, thus removing the main bottleneck to the strong-scalability of CFD.

The novel chaotic solver and chaotic-cycle multigrid methods have been implemented as an open-source library, *Chaos*. It is hoped that work on these scalable solvers can be continued and applied to other disciplines.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, James Hawkes, declare that this thesis entitled *Chaotic Methods for the Strong Scalability of CFD* and the work presented in this thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [36, 37, 38, 39, 40].

Date: 10/08/2017

Signed:

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisors: Professor Stephen Turnock, Dr. Guilherme Vaz, Professor Simon Cox and Dr. Alexander Phillips – for their support, patience, flexibility and invaluable advice. It has been a joy to conduct research under their guidance.

In addition, I wish to thank the Maritime Research Institute Netherlands (MARIN), for accommodating me in their Netherlands office for 15 months and funding this PhD. In particular, my thanks goes to Dr. Guilherme Vaz, Dr. Christiaan Klaij, and other members of the ReFRESCO development team. Without their immense technical expertise, my research would not have been possible. Furthermore, I wish to thank Dr. Guilherme Rosetti, whom I had the opportunity to intern with, at MARIN, prior to this PhD.

I wish to thank Dr. Andrew Gerber and the Envenio team, of the University of New Brunswick (UNB). Three months were spent working in their offices in Eastern Canada, during which time a huge amount of knowledge was shared. In particular, I enjoyed discussing multigrid methods (and playing chess) with Araz Eghbal.

Thousands of simulations have been performed to obtain the results presented in this thesis, and thousands more during development and preliminary investigations. I wish to extend my gratitude to the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work. In particular, I wish to thank Ivan Wolton for his cooperation in running the large 1024/2048-core jobs on Iridis4 – without which many meaningful results would have been unobtainable.

I would like to thank my parents for their unwavering support and inspiration, throughout this PhD and life in general. Finally, thank you to my wife, Kimberley. Her patience, especially during my extended trips to the Netherlands and Canada, has been invaluable.

Nomenclature

Symbols may be reused on occasion, to remain consistent with literature. Other symbols and abbreviations may appear with limited scope.

Symbols

$\mathbf{A}$	The coefficient matrix of a linear equation-system
a_{ij}	Component of $\mathbf{A}$
a_{max}	Maximum multigrid aggregation size
$\mathbf{b}$	The right-hand side of a linear equation-system
b_i	Component of $\mathbf{b}$
C	Number of cores
C_f	Coefficient of friction
C_p	Coefficient of pressure
$\mathbf{D}$	The diagonal of a coefficient matrix $\mathbf{A}$
i, j	Generic indices of a matrix/vector/grid/loop
k	Iteration counter for an iterative linear solver
$\mathbf{L}$	The lower triangle of a coefficient matrix $\mathbf{A}$
L	Length [m]
$\mathbf{M}$	The iteration matrix of a linear equation-system solver; usually the Jacobi iteration matrix
m	Multigrid level
N	The number of elements in a CFD simulation, or rows in a linear equation-system
n	The number of elements in a CFD simulation, or rows in a linear equation-system, local to a single process
$\mathbf{P}$	Multigrid prolongation matrix
p	Number of pre- and post-smoothing iterations in a multigrid method
$\mathbf{R}$	Multigrid restriction matrix
$\mathbf{r}$	The residual vector of a linear equation-system

r_i	Component of $\mathbf{r}$
Re	Reynolds Number
S	Scalability factor on C cores, normalized to performance on X cores, given by $= XT_x/T_C$
s	Asynchronicity of a chaotic relaxation
t	Iteration counter for the SIMPLE time-step loop
T_C	Absolute wall-time consumed by a program or routine on C cores [s]
$\mathbf{U}$	The upper triangle of a coefficient matrix $\mathbf{A}$
V	Velocity [m/s]
w	Iteration counter for the SIMPLE outer loop
$\mathbf{x}$	The solution vector of a linear equation-system
x_i	Component of $\mathbf{x}$
α	Under- or over-relaxation factor of the outer loop
ω	Under- or over-relaxation factor of a linear iterative solver
$\rho(\cdot)$	Spectral radius of a matrix
$_A$	Subscript denoting the process-local block of a vector or matrix
$_B$	Subscript denoting the off-process portion of a vector; or local portion of a matrix which should be multiplied by said vector portion

Abbreviations

BCGS	Bi-Conjugate Gradient Squared method
CFD	Computational Fluid Dynamics: the practice of simulating fluid flows using numerical methods
CPU	Central Processing Unit: a single processing chip or processor, often consisting of many cores
FLOPS	Floating-Point Operations per Second: often given with a unit prefix (*i.e.* TFLOPS, PFLOPS, EFLOPS)
GMRES	Generalized Minimal Residual method
HPC	High-Performance Computing: the practice of using supercomputers to perform large-scale computations
I/O	Input/Output, typically referring to non-volatile data storage (*i.e.* hard drive storage)
ILCT	Inner-Loop Convergence Tolerance: the relative convergence tolerance of individual linear equation-systems in the SIMPLE algorithm
KSP	Krylov-subspace methods for the solution of linear equation systems
LES	Large-Eddy Simulation

LLC	Last-Level Cache: the largest cache register on a chip, typically shared between cores
MARIN	Maritime Research Institute Netherlands
MPI	Message Passing Interface: a standardized, portable message-passing interface, enabling communication between processes
OpenMP	Open Multi-Processing: a standardized API enabling multi-threading within a process, using shared memory
PETSc	Portable, Extensible Toolkit for Scientific Computation: includes a range of linear equation-system solvers
QUICK	Quadratic Upstream Interpolation for Convective Kinematics: a 2^{nd} order discretization scheme
RAM	Random-Access Memory: volatile off-process memory accessible by all CPUs on a node
RANS	Reynold's Averaged Navier-Stokes Equations
ReFRESCO	Reliable & Fast RANS Equations solver for Ships, Cavitation and Offshore
SIMD	Single-Instruction Multiple-Data: a fine level of intra-core parallelization based on vectorization
SIMPLE	Semi-Implicit Method for Pressure-Linked Equations
SOR	Successive Over-Relaxation
SpMV	Sparse-Matrix Vector multiplication

Definitions

Accelerator	Generic term for any co-processor, GPU, or other device, designed to attach to a node and supplement the capabilities of the main CPU(s)
Cache	Small on-chip memory for fast storage of frequently-used data and instructions
Core	An individual processing unit, forming part of a processor, capable of independently reading and executing program instructions
Inner Loop	The iterative loop performed to solve each linearized equation-system in the SIMPLE algorithm
Node	Resembles an independent computer, with dedicated CPU(s) and (usually) RAM; many nodes are joined together to create a supercomputer
Outer Loop	The iterative loop performed by the SIMPLE algorithm, to update non-linear quantities and couple equations; synonymous with 'SIMPLE loop' or 'non-linear loop'

Process	An execution space for a program, with dedicated memory which is not shared with other processes; a message-passing interface is often used to communicate between parallel processes
Supercomputer	A group of independent nodes connected with an inter-nodal network to create a large parallel computer
Thread	An entity within a process, which shares memory of said process, and is scheduled and managed by the process – typically using a multi-threading API such as OpenMP

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is the science of modelling fluid flows and heat transfer using numerical techniques. The equations that describe Mother Nature herself are approximated and solved, creating simulations which provide important physical results – often as an alternative to expensive experimental testing and prototyping. CFD can be used to model dynamics around aircraft, ships and land vehicles; and also has uses in engine design, civil engineering, weather forecasting, blood-flow prediction, computer-generated imagery (CGI) and much more (see figure 1.1). The research and development presented herein is particularly focused on the maritime industry, but is transferable to many applications of CFD.

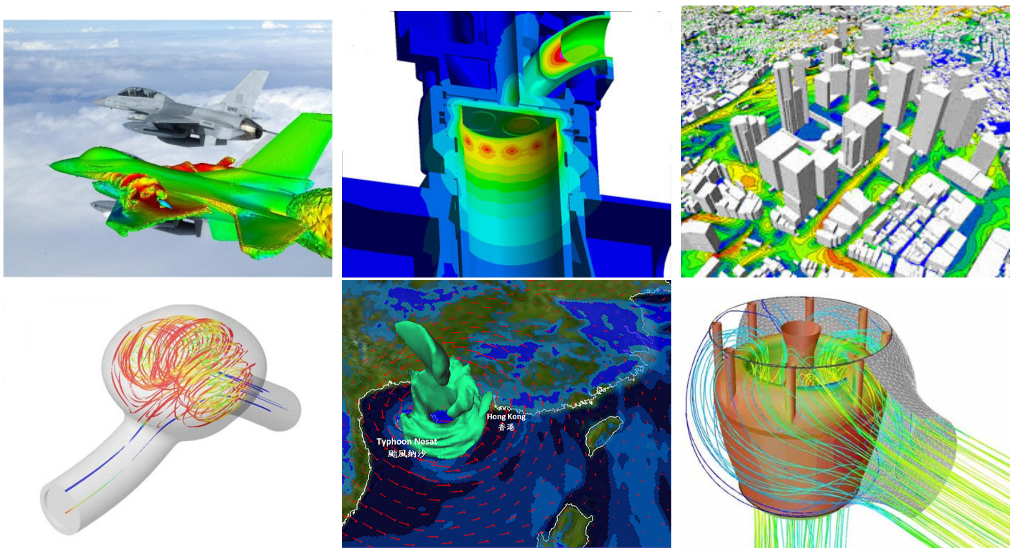


Figure 1.1: Examples of CFD applications. Clockwise from top left: aircraft simulation; flow and heat transfer within an internal combustion engine; prediction of pedestrian wind environment around buildings; computed flow through a cerebral aneurysm; severe weather prediction; coolant flow through a pump, used to cool a nuclear reactor core (cfd4aircraft.com, ec.europa.eu, flometrics.com, htf.fl, hko.gov.hk, aij.or.jp).

In the maritime industry, CFD simulations are often used to estimate power requirements for vessels; to aid the design of propellers; to predict seakeeping performance; and to determine manoeuvring characteristics (see figure 1.2). As new features are added to state-of-the-art codes, CFD becomes an increasingly useful tool to assist in the design of maritime vessels, offshore structures and renewable-energy infrastructure.

To enable such rich simulations, CFD codes are often run on high-performance computing (HPC) facilities, using supercomputers which consist of many parallel computing nodes. As the compute power from these supercomputers grows, with technological advancements in both hardware and underlying algorithms, CFD simulations can be performed with higher resolution and at higher speeds, improving accuracy and turnaround time. To make use of these supercomputers, the CFD algorithm is parallelized – that is, the simulation is split into spatial partitions which can be run on individual nodes within a supercomputer, with some communication overhead.

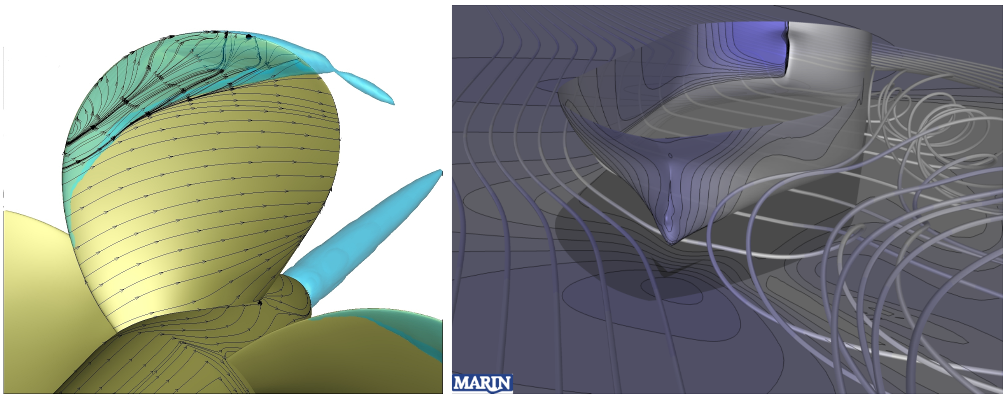


Figure 1.2: Examples of maritime CFD simulations. A cavitating propeller [left] and a vessel undergoing manoeuvring trials [right] (refresco.org).

Over the last few decades, growth in supercomputing power has been exponential, with floating-point-operations-per-second (FLOPS) doubling approximately every 13.85 months [96]. Whilst this growth is relatively constant, the underlying architectures which achieve such growth are not. Until 2004, the speed and electricity consumption of transistors was governed by Dennard's Scaling: as transistors shrank in size, their speed increased linearly and their electricity consumption dropped quadratically [23]. Unfortunately, the transistors used in modern processors are so small that electrons are able to 'leak' across the dielectric gates, and voltages must be increased to maintain stability. This limitation, known as the 'power wall', has stopped the exponential growth of core clock speeds – and in order to continue exponential scaling of supercomputers, manufacturers must adapt. Since 2004, 'multi-core' machines have become commonplace: packaging multiple 'slow' cores into a single processor in order to continue an exponential increase in FLOPS-per-node.

Moore's law [28], which predicted that transistors would halve in size every 18 months, underestimates the exponential growth in supercomputing power (doubling every 13.85 months), because supercomputers capitalize on other hardware developments, such as improvements to inter-nodal network and simple scaling of the number of nodes in a system. Unfortunately, these technologies are also slowing due to other electrical efficiency constraints [54]. For example, inter-nodal scaling reached limitations due to the simple cost of transmitting bytes of data through a copper wire – and developments into optical interconnects have not provided exponential improvements.

By 2020, the first *exascale* supercomputer is expected, capable of 10^{18} FLOPS (or 1 ExaFLOPS), but it is heavily limited by electrical efficiency constraints. To put this electrical efficiency problem into perspective, if current multi-core architectures were scaled to exascale capability, they would require 60MW of electrical power (approximately $60-million US dollars per year at 2014 rates)[85]. Currently[1], the most powerful computer in the world, *Sunway TaihuLight*, consumes 15.3MW of electrical power (US$15.3-million/year) to produce 93 PFLOPS of compute power – it cost US$273-million to build.

The only remaining means of efficiently scaling FLOPS, and maintaining supercomputer growth, within a reasonable power envelope, is to increase the number of cores-per-node exponentially. The multi-core era is ending, and a new era, dubbed the 'many-core' era has begun, where hundreds or thousands of cores may be packaged into a single node to counteract stagnated core-clock speeds and inter-nodal scaling.

The many-core era is already established at the cutting edge of supercomputer development, although there are multiple competing many-core architectures. *Sunway TaihuLight* (#1 of the Top500) uses 260-core processors to achieve a peak performance of 93 PFLOPS with a power consumption of 15.3MW. *Tianhe-2* (#2) uses 3 Xeon Phi co-processors to accelerate each node, providing 192 cores-per-node (totalling 34 PFLOPS at 17.8MW). The #3 supercomputer, achieving 17.6 PFLOPS (at 8.2MW), uses k20 graphics cards providing 2496 low-speed cores per node. The Argonne Leadership Computing Facility (ALCF) have recently commissioned a new supercomputer, *Aurora*, to be complete in 2018 [4]. It is likely to be the most powerful supercomputer in the world upon completion, with a projected performance of 180 PFLOPS and an electrical power consumption of just 13MW – see figure 1.3. *Aurora* follows the predicted trends for next-generation supercomputing:

> *"Aurora will deliver more than eighteen times the computational performance of Mira, its predecessor at the ALCF, using a nearly equal number of compute nodes. Aurora will be a many-core system but with nearly an order of magnitude more [Xeon Phi] processors."*
>
> – ALCF, aurora.alcf.anl.gov

Whilst these machines are the highest-calibre, largest-scale supercomputers available, the hardware which they use is not exclusive to these machines. No matter the size of machine which individual institutions acquire, their hardware is usually similar, in order to take advantage of all the technological improvements (such as decreased electricity consumption) that are available – although these technologies do take some time to trickle down to the industrial or academic level.

CFD has benefited from good *weak-scalability* for many years. Good weak scalability means that if the number of mesh cells and computational cores grow at the

[1]Supercomputing statistics in this thesis are based on the Top500 list published in November 2016 [96].

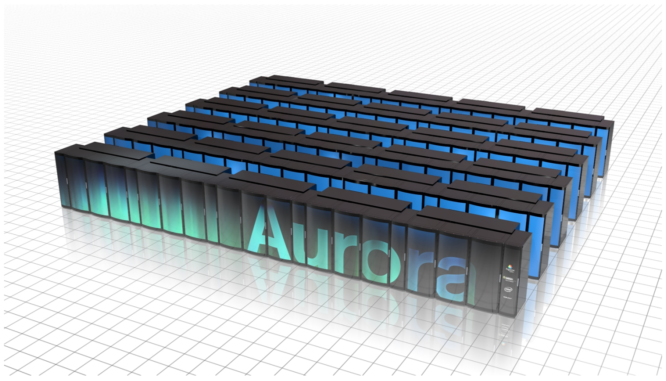


Figure 1.3: Artists rendering of *Aurora*, commissioned by the Argonne Leadership Computing Facility (ALCF) for 2018, will be capable of 180 PFLOPS. [intel.com]

same rate (thus the *cells-per-core* ratio stays the same), then the computational time remains more-or-less constant. Below a certain cells-per-core ratio, efficiency losses become overwhelming as parallel-overhead in the CFD algorithms becomes large. The critical ratio depends heavily on the code, computing facility and simulation specifics. Maintaining this ratio, and thus maintaining weak-scalability, has required little in the way of algorithmic changes for CFD.

An exascale supercomputer is predicted to contain between 1-billion and 10-billion cores, by 2020; an increase of approximately 100-1000$\times$ compared to *Sunway TaihuLight*, for just a 10.75$\times$ increase in FLOPS. If one assumes that the size of CFD simulations increases proportionally with total FLOPS, then the cells-per-core ratio will decrease by a factor of 10-100 in the next 4 years.

This translates to more practical levels too. *Iridis4* is the University of Southampton's multi-core supercomputer, built in 2013 and capable of 250 TFLOPS, using 770 16-core nodes (for a total of 12,320 cores). If one makes the same analysis, again assuming that maximum CFD simulation sizes are proportional to total FLOPS, a numerical code that performs well on Iridis4 must maintain this performance when the cells-per-core ratio is reduced by a factor of 20-200, in order to perform well at exascale.

This abrupt increase in parallelization, and reduction in cells-per-core ratio, means that the weak scalability of CFD is no longer relevant. In order for CFD to increase its capabilities, and even just to remain competitive, it is important to consider *strong scalability*: the ability to achieve an X-times speed-up when X-times more cores are employed, with a constant problem size. In other words, the ability to reduce the cells-per-core ratio without loss of efficiency must be improved.

The nature of simulations at the forefront of maritime CFD capability are also changing. Most state-of-the-art simulations are unsteady (time-dependent) requiring CFD algorithms to simulate multiple time-steps of a flow field. The three spatial dimensions can be parallelized, by partitioning the domain, but the time dimension is strictly serial (excepting new parallel-in-time methods [24]) – requiring each previous time-step to be computed before another can begin. To improve the capabilities of these unsteady simulations, by bringing their simulation cost down to reasonable levels, the spatial dimensions must be parallelized further in order to accelerate each time-step.

Thus, another motivation for efficiently reducing the cells-per-core ratio.

To summarize, changes in computing architecture and emphasis on unsteady simulations place pressure on CFD codes to adapt. Particularly, emphasis must be placed on improving the strong scalability of existing codes and their underlying algorithms, in order to enable feature-rich, high-resolution, accurate simulations utilizing many-core architectures.

1.1 Research Question

Inefficient utilization of many-core computing architectures limits the capabilities of computational fluid dynamics, thus driving the research question for this PhD:

> *"What limits the strong scalability of CFD and its ability to handle many-core architectures? What can be done to improve the CFD algorithms in this respect?"*

1.2 Aim

The aim of this PhD is to prepare CFD for the many-core era. Using *Iridis4* and a state-of-the-art CFD code, *ReFRESCO*, an in-depth investigation of the strong scalability of CFD will be performed. Following this, the principle bottlenecks to scalability will be examined in more detail – attempting to find or develop novel methods to circumvent these bottlenecks and unlock the potential of massively-parallel supercomputing.

1.3 Objectives

The objectives of this PhD are incremental, with the results of each study guiding the goals for further work. This process is explained below:

- **Research into the current state and future trends of CFD and HPC.** The first objective is to thoroughly examine the CFD and HPC environment: firstly considering the current state-of-the-art and core technologies; and secondly, identifying ways in which the HPC ecosystem is changing, and ways in which the CFD algorithm is likely to change, over the next few years. The knowledge gained from this objective will define the main scalability concerns, highlight opportunities for improvements, and ensure that any work carried out during this PhD is compatible with other developments in the CFD domain.
- **Perform scalability experiments to determine the bottlenecks to strong-scalability of CFD.** The next objective is to measure the current state of CFD in terms of strong scalability. Advanced software-profiling and extensive testing on a large number of computational nodes and a variety of test cases can be used to provide insight into the code – complementing and extending existing literature. By intelligently partitioning the CFD algorithm into key routines, it is possible to identify which areas require further development. As will be seen,

the results of this study show that overall scalability suffers due to the linear equation-system solver for the Poisson pressure equation, thus motivating research into the scalability of linear solvers.

- **Investigate state-of-the-art methods to circumvent strong-scalability bottlenecks.** Here, the objective is to determine whether the previously-identified scalability bottlenecks can be bypassed using state-of-the-art algorithms. A wide variety of linear equation-system solvers and preconditioners are tested; particularly those which are designed to improve strong scalability. Although the results showed some improvements from these newer solvers, the bottlenecks are not adequately removed. Thus, a further outcome from this objective is to determine the shortcomings of these methods, and identify the reasons for their poor scalability. One of the main reasons identified is excessive global synchronization, caused by communications between processes.

- **Develop and test a linear equation-system solver designed for strong scalability, based on the theory of Chaotic Relaxation.** The theory of Chaotic Relaxations [16] is an interesting foundation for the development of a highly-scalable 'chaotic solver', which could provide fast approximations to the pressure equation by avoiding all forms of synchronization. The objective is to develop this solver, verify the implementation, and measure performance and scalability against other state-of-the-art solvers. The results showed good scalability, but poor numerical performance compared to state-of-the-art solvers, thus motivating the next objective.

- **Develop and test a novel 'chaotic-cycle' multigrid method, combining chaotic relaxation theory with algebraic multigrid solvers.** Earlier investigation showed that the scalability of algebraic multigrid methods is poor; but there is potential to combine chaotic relaxation theory and multigrid methods, building on the chaotic solver developed previously. The objective is to develop, verify and test the performance of a multigrid method using a novel 'chaotic-cycle' algorithm. The overall goal is to determine whether these methods can replace existing linear-solvers and allow CFD to scale efficiently on foreseeable computing architectures.

- **Assessing the suitability of chaotic solvers and multigrid methods for practical CFD test cases.** This final objective seeks to determine whether the chaotic methods developed in this thesis can be applied safely to practical CFD problems. In particular, it should be verified that the non-deterministic, chaotic nature of these solvers does not negatively affect the overall flow solution, especially when the flow is unsteady and takes several days to compute (with thousands of independent calls to the chaotic methods). Larger grids are used for these tests, which also provides an opportunity to verify the $O(N)$ performance of the chaotic-cycle multigrid method and observe some scalability characteristics on larger grids.

1.4 Publications

The following publications have been produced during this PhD. The following information is correct at the time of writing:

- **Performance Analysis Of Massively-Parallel Computational Fluid Dynamics**, presented at *The 11th International Conference on Hydrodynamics (ICHD)* in 2014 [36]. Appendix A.
- **Potential of Chaotic Iterative Solvers for CFD**, presented at *The 17th Numerical Towing Tank Symposium (NuTTS)* in 2014 [37]. Appendix B.
- **Chaotic Linear Equation-System Solvers for Unsteady CFD**, presented at *The 6th International Conference on Computational Methods in Marine Engineering (MARINE)* in 2015 [38]. Appendix C.
- **On the Strong Scalability of Maritime CFD**, published in the *Journal of Maritime Science and Technology* in 2017 [39]. Appendix D.
- **Chaotic Multigrid Methods for the Solution of Poisson Equations**, submitted to the *Journal of Computational Physics* in 2017 [40]. Under review. Appendix E.

A further paper has been submitted as co-author, **Modelling Transition using Turbulence and Transition Models for a Flat Plate Flow: Uncertainty, Verification and Sensitivity** [81], studying laminar-turbulent transition models in ReFRESCO; though this is largely unrelated to the thesis.

A bitbucket repository for the open-source *Chaos* library, published as part of this PhD, is publicly available at `https://bitbucket.org/jamesnhawkes/chaos`.

1.5 Thesis Outline

This thesis begins with a preliminary background chapter (2), which covers the fundamentals of Computational Fluid Dynamics (CFD) and High-Performance Computing (HPC). This background chapter also serves to describe the semi-implicit, viscous-flow, CFD code used during this thesis (ReFRESCO) and the University of Southampton supercomputer (Iridis4). In the following chapter (3), literature is reviewed which focuses on the future of CFD and HPC, and methods for dealing with the changes to each – including changes in programming paradigms and numerical algorithms. The intent of these two chapters is to show the present and future, respectively, of CFD and HPC, and reinforce the aim of this PhD. This also covers the first objective described above. Each of the subsequent objectives are presented in the following chapters, introducing their own background and literature where necessary.

Chapter 4 focuses on an initial strong scalability assessment of ReFRESCO, in which software profiling tools are used to break down the algorithm and measure scalability. This portion of the PhD provided unique research to the field by carrying out a detailed scalability assessment, whilst considering the effects of multiple user-settings (such as discretization techniques, modelling techniques, convergence tolerances, choice of linear equation-system solvers, and mesh structure). Following this, chapter 5 investigates, in more detail, the effects of alternative linear solvers and preconditioners on the performance and scalability of the code. These two chapters thoroughly identify the bottlenecks to massively-parallel CFD, and the reasons for their poor scalability.

Chapter 6 covers the development and testing of a *chaotic solver*, based on the theory of chaotic relaxation [16]. Chaotic solvers have been tested in literature and show good results in terms of scalability – but only compared to their non-chaotic counterparts (Jacobi and Gauss-Seidel). Practical comparisons to state-of-the-art solvers (such as those investigated in earlier chapters) could not be found in the literature; and the application of chaotic solvers to CFD had also not been considered. In this chapter, a novel type of chaotic solver is implemented and tested via ReFRESCO, and it's performance compared to a large range of state-of-the-art solvers. An open-source library, *Chaos*, which encapsulates all of the novel developments of this PhD, is also introduced in this chapter.

Overall, the scalability of the chaotic solvers is impressive, but their raw numerical performance is not competitive with modern solvers, especially as the size of the CFD simulation increases. However, applying the same chaotic theories to an algebraic multigrid method may provide good scalability without any sacrifice to numerical performance. Chapter 7 covers the design of a novel 'chaotic-cycle' multigrid, including an in-depth discussion of the various nuances and intricacies of the algorithm. The chaotic-cycle multigrid has been developed in *Chaos* and tested against various other solvers with good results, although there is still scope for improvement.

In chapter 8, the chaotic solver and chaotic-cycle multigrid are used on a modern, relevant, maritime CFD test case – the BB2 submarine [97]. The intent of this study is to: (a) show that the flow solution generated by non-deterministic solvers is as good as that from deterministic solvers; (b) show that the performance of the chaotic-cycle multigrid method scales correctly (linearly) with the size of the CFD simulation; and (c) investigate whether unsteadiness (time-dependence) of the CFD simulation affects the performance of the chaotic and non-chaotic solvers. The submarine is simulated at a 20° angle of drift, and is highly unsteady. In reality, this simulation should be carried out at much higher resolution, but the purpose here is not to create a perfect physical solution – but to demonstrate the overall correctness of chaotic solvers in the context of practical CFD. These last two chapters aim to prove the performance and suitability, respectively, of the chaotic-cycle multigrid method for massively-parallel CFD.

Chapter 9 (Conclusions), discusses the successes and short-comings of the work carried out during this PhD, and assesses the extent to which the research question was answered. It also frames the novel contributions of this work in the wider context of CFD and numerical science in general. Chapter 10 identifies areas of further work and research, including improvements to the performance of *Chaos* and the chaotic-cycle multigrid algorithm. Finally, the aforementioned publications are included as appendices.

Chapter 2

Background

Here, a brief background to CFD and HPC is given. The intention is to highlight and explain those aspects of each which are relevant for this thesis. The CFD code (*ReFRESCO*) and supercomputer (*Iridis4*) used throughout this PhD are also introduced.

2.1 Computational Fluid Dynamics (CFD)

There are a huge variety of CFD methodologies, which cannot all be discussed here. Instead, the methodology employed by ReFRESCO is described, since it is most relevant to this thesis. Much of the following derivation is interpreted from Kuzmin [55], and is favoured for its succinctness. Alternative approaches can be found in Ferziger & Perić [26] or Murthy & Mathur [67]. Ferziger & Perić illustrates the integral form of the equations which are used here, since they are most relevant for the finite-volume method used by most hydrodynamic CFD codes – including ReFRESCO.

2.1.1 Fundamentals

The governing equations for CFD are derived by considering the quantity of any general variable, ϕ, within a fluid domain, V (figure 2.1). ρ is the density of the variable, Γ is a diffusion coefficient matrix and q_ϕ represents production (sources & sinks) of ϕ. S is the enclosing surface of V with normal vector $\mathbf{n}$. $\mathbf{u}$ is the velocity vector of the fluid.

$$\underbrace{\frac{\partial}{\partial t}\int_V \rho\phi \,\mathrm{d}V}_{\text{Transient}} + \underbrace{\int_S (\rho\phi\mathbf{u})\cdot\mathbf{n}\,\mathrm{d}s}_{\text{Convection}} - \underbrace{\int_S (\Gamma\nabla\rho\phi)\cdot\mathbf{n}\,\mathrm{d}s}_{\text{Diffusion}} = \underbrace{\int_V q_\phi \,\mathrm{d}V}_{\text{Production}} \tag{2.1}$$

When multiple control volumes are considered in a mesh, this equation can be used to find the transport of a variable through a large domain. Hence this equation is often referred to as the general transport equation – its various components representing the different means of transporting some variable. If each cell, c, in a mesh, has f faces – each shared between exactly two cells, then one can approximate the total flow in a domain by discretizing the general transport equation, and applying it to each cell (figure 2.2). In this example (equation 2.2), a backwards-differencing scheme is used to

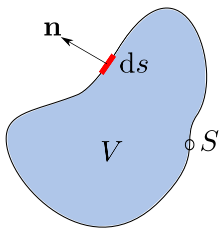


Figure 2.1: A diagram of an arbitrary fluid domain V, with surface S.

discretize the transient term. V_c denotes the volume of each cell, $\mathbf{S}_f$ is the area-vector of each surface $(A \cdot \mathbf{n})$ and subscript f and p represent where the variable is evaluated – at the face-centre or cell-centre respectively:

$$\underbrace{\frac{\frac{3}{2}\phi_p^n - 2\phi_p^{n-1} + \frac{1}{2}\phi_p^{n-2}}{\Delta t}\rho V_p}_{\text{Transient}} + \underbrace{\sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f}_{\text{Convection}} - \underbrace{\sum_f \mathbf{S}_f \cdot (\rho \Gamma \nabla \phi)_f}_{\text{Diffusion}} = \underbrace{s_{\phi_p} V_p}_{\text{Sources}} \tag{2.2}$$

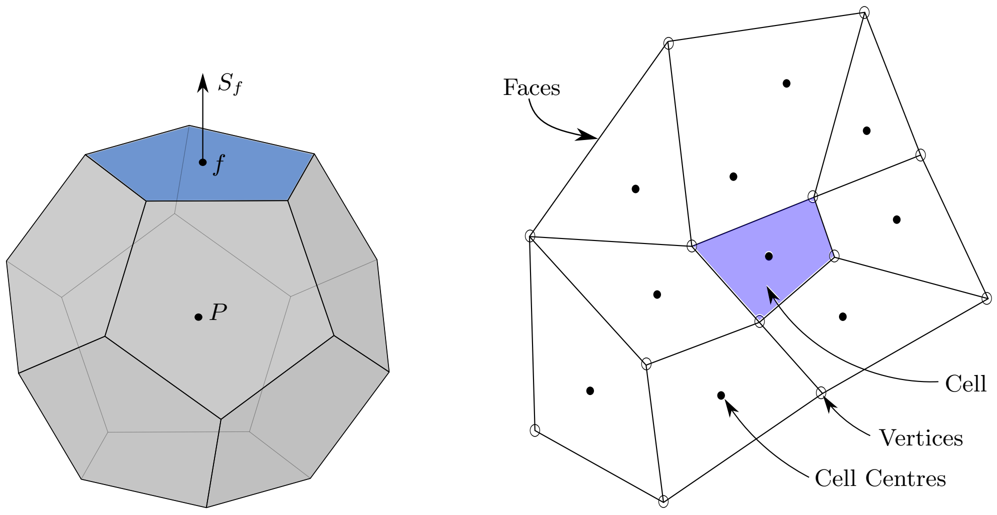


Figure 2.2: [Left] A single three-dimensional cell and [right] a diagram of a small two-dimensional mesh.

An approximation due to the discretization of the transient term has already been introduced. Further approximation may be introduced through interpolation or geometric approximation, which are required to solve the discretized general transport equation.

Firstly, we consider *interpolation* of face-centred values from cell-centred values. Note that the convection term requires face-centred values of ϕ_f, but ϕ is only computed at cell centres. Interpolation between two (or more) neighbouring cell-centres is required to find a face-centred value. The simplest scheme is the central differencing scheme, which takes a weighted average of the neighbouring cell-centred values to produce a second-order approximation. It frequently suffers from non-physical oscillations of transported variables. A common first-order scheme is the upwind scheme which approximates the face-centred value by taking only the cell-centred value from the cell upstream. A blend of the two schemes is often used for reasonable accuracy and stability. Higher-order upwind methods such as QUICK (Quadratic Upwind Interpolation for Convective Kinematics) are often used for greater accuracy. Interpolation is also required to obtain face-centred gradients $\nabla\phi_f$ in the diffusion term. Central differencing suffices in this case.

Note that the cell-centred gradients $\nabla\phi_c$ are found by applying Gauss theorem with the just-interpolated face values of ϕ_f, thus the diffusive term suffers from any interpolative inaccuracy in the convection term.

$$\int\limits_{V_p} \nabla\phi \, \mathrm{d}V_p = \int\limits_{S} \phi \cdot \mathbf{n} ds, \quad \therefore (\nabla\phi)_p = \frac{1}{V_p} \sum_f \mathbf{S}_f \phi_f \tag{2.3}$$

These interpolations are approximations, made worse by highly eccentric or non-orthogonal meshes. Eccentricity causes problems for convection approximations, since the face-centre no longer lies between the cell-centres. Non-orthogonality causes problems for diffusion approximations, where the face-normal vector is no longer parallel to cell-centre vector (figure 2.3).

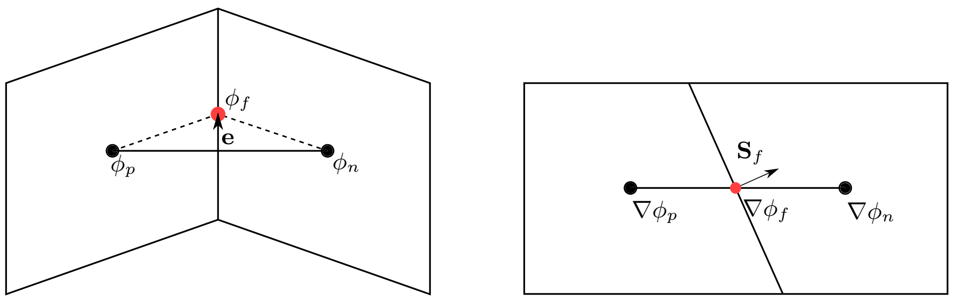


Figure 2.3: [Left] A diagram illustrating eccentricity problems for convective interpolation and [right] non-orthogonality problems for diffusive interpolation.

Secondly, *geometric approximation* can be introduced through non-planar surfaces. The area and normal vector of a face can be calculated by splitting the face into triangles, summing the areas and averaging the normal vectors. In some cases, the normal vectors are not parallel and an approximation of a twisted face to a planar face is applied. This has repercussions for the computation of cell volumes, which is performed via Gauss theorem.

$$\int\limits_{V_p} \mathrm{d}V_p = \int\limits_{S} \mathbf{n} ds, \quad \therefore V_p = \sum_f \mathbf{S}_f \tag{2.4}$$

Momentum Equations

In the discretized general transport equation, ϕ can be replaced by any quantity of interest. First and foremost, it can be used to find the momentum of the flow itself, in the $x-$, $y-$ and $z-$ dimensions. In this case $\phi = \mathbf{u}$, the diffusion term vanishes ($\Gamma = 0$), and the equation takes the general form $ma = \Sigma F$ (Newton's Second Law), where the source term q_ϕ has become the sum of forces acting on the cell volume. The expression for production of ϕ in the general transport equation assumes sources of momentum can be expressed as body forces (acting on the volume) – which is true for gravitational, electromagnetic, Coriolis or centrifugal forces. However, momentum is also produced through surface shear stresses, due to fluid viscosity, which must be captured. For Newtonian fluids it is possible to express the shear stresses τ in terms of strain rate:

$$\tau = \mu(\nabla \mathbf{u} + (\nabla \mathbf{u})^T) - \frac{2}{3}\delta\mu\nabla \cdot \mathbf{u} \tag{2.5}$$

where δ is the Kronecker delta and superscript ^{T} represents the transpose. This has a similar format to the missing diffusion term, so shear stress (τ) and pressure ($-p$) can be combined, yielding the following:

$$\begin{split}\frac{\frac{3}{2}\mathbf{u}_p^n - 2\mathbf{u}_p^{n-1} + \frac{1}{2}\mathbf{u}_p^{n-2}}{\Delta t}\rho V_p + \sum_f \mathbf{S}_f \cdot (\rho \mathbf{u}\mathbf{u})_f \\ - \sum_f \mathbf{S}_f \cdot \Big(\mu(\nabla \mathbf{u} + (\nabla \mathbf{u})^T) - \delta(p + \frac{2}{3}\mu\nabla \cdot \mathbf{u})\Big)_f = F_b V_p,\end{split} \tag{2.6}$$

where F_b is the sum of all body forces evaluated at the cell centre.

Continuity Equation

To close this system, the pressure term which has just be introduced must also be transported. In this case $\phi \equiv 1$, $q_\phi = 0$, and $\Gamma = 0$. For a fluid with low Mach number (< 0.3) density can be considered constant, thus the transient term also vanishes.

$$\sum_f \mathbf{S}_f \cdot \mathbf{u}_f = 0. \tag{2.7}$$

Equations 2.6 and 2.7 are the Navier-Stokes equations, in their discrete form. They create a coupled, non-linear system that can be solved to simulate unsteady (time-dependent) fluid flows within a domain. To fully resolve these equations, the discrete mesh must be fine enough to resolve the smallest scales of turbulence in the flow. For most practical cases this is impossible, and techniques are used to model smaller-scale motions of the flow, instead of simulating them directly.

A common approach is to use Reynolds Averaging, whereby the velocity $\mathbf{u}$ is split into its mean value and turbulent fluctuations, $\bar{\mathbf{u}} + \mathbf{u}'$. The Navier-Stokes equations are then time-averaged to produce the Reynold's Averaged Navier-Stokes (RANS) equations. In the linear terms of the RANS equations: $\overline{\bar{\mathbf{u}} + \mathbf{u}'} = \bar{\mathbf{u}}$; the mean of the fluctuations is simply zero. However, in the non-linear terms appearing in the momentum equations: $\overline{(\bar{\mathbf{u}} + \mathbf{u}')(\bar{\mathbf{u}} + \mathbf{u}')} = \overline{\bar{\mathbf{u}}\bar{\mathbf{u}}} + \overline{\mathbf{u}'\mathbf{u}'}$; the mean of the squared fluctuations cannot be evaluated to zero. Thus an additional item ($\rho\mathbf{u}'\mathbf{u}'$) has been created in the convective term.

This part of the equation is usually shifted into the diffusive term, with the notation $\tau_r = -\rho\overline{\mathbf{u}'\mathbf{u}'}$. In this format, it is similar to a stress tensor, and hence τ_r is called the Reynold's-stress tensor.

The Boussinesq hypothesis allows a relation between the Reynold's stresses and the mean velocity gradients, thus:

$$\tau_r = -\rho\overline{\mathbf{u}'\mathbf{u}'} = \mu_T(\nabla\bar{\mathbf{u}} + (\nabla\bar{\mathbf{u}})^T) - \delta\frac{2}{3}\rho k \tag{2.8}$$

where μ_T is the *eddy viscosity* (a flow-dependent equivalent to molecular viscosity μ), δ is the Kronecker delta, and $k = \frac{1}{2}\overline{\mathbf{u}' \cdot \mathbf{u}'}$ is the turbulent kinetic energy. These terms are then modelled via additional transport equations for relevant turbulent variables.

A common turbulence model, the Spalart-Allmaras model [89], is a one-equation model that transports μ_T directly and ignores k. A variety of two-equation models exist, such as the commonly-used $k - \omega$ SST-2003 model [63], which transports the kinetic energy and dissipation rate ω, granting:

$$\mu_T = \frac{\rho k}{\omega} \tag{2.9}$$

Turbulence can also be modelled in other ways. Large-Eddy Simulations (LES), as the name suggests, simulates the larger turbulent structures but models the smaller, geometry-independent structures. The equations are similar, although low-pass filtering is used instead of Reynold's averaging; and special turbulence models are used which only model sub-grid-scale structures. In all of these cases, the modelled turbulence quantities are transported using the general form of the transport equation (equation 2.6).

For specialist flows, as appearing in maritime CFD, other flow features are often modelled and their variables transported via additional transport equations. Commonly, equations are used to create homogeneous two-phase flows (i.e. for modelling air-water interfaces, where each fluid is considered constant-density). For free-surface type flows, a volume-fraction variable is modelled and transported, representing the proportions of air/water in any given cell. For some maritime flows, cavitation is important; and similar equations are used to model the production, growth and collapse of underwater air bubbles.

2.1.2 Solution Methodologies

The previous section has shown how the flow of a fluid within a domain can be approximated, simulated and modelled via the RANS equations and various additional transport equations, by applying them at each point in a discretized mesh. For a typical maritime application, there may be many non-linear, tightly coupled partial differential equations to solve; including those for momentum, continuity, turbulence, free-surface and cavitation modelling.

The additional transport equations depend on the solution of the momentum and continuity equations – but also directly feed back into them via Reynold's Stresses or through additional production terms in the Navier-Stokes equations. All the while, the momentum equations are coupled together; and the continuity equation must be satisfied.

In this section, the methodology for solving this complex system of equations will be introduced. The CFD code being used for this thesis, ReFRESCO, uses the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm; and this will be explored here. Other options for solving this system include PISO (Pressure Implicit with Splitting of Operators), SIMPLER (SIMPLE-Revised) and SIMPLEC (SIMPLE-Corrected) algorithms. The fundamentals of all the methods are similar [26].

The first principle is that of linearization, performed using a Picard's method, whereby the system of non-linear PDEs are converted to their linear counterparts which can be solved iteratively to achieve a correct solution. Consider a loop with counter w, which shall be called the *non-linear* or *outer* loop. Any non-linear terms in the RANS equations are replaced by a linear equivalent, by taking higher-order components from the previous loop, $w-1$. For example, the term $\overline{\mathbf{uu}}$ can be replaced by $\overline{\mathbf{u^w u^{w-1}}}$. A seed for the flow field at $w=0$ must be provided, typically set to a uniform flow-field or using some pre-supposed flow information.

At each non-linear iteration, w, every equation must be solved and coupled; but the equations are now linear PDEs which means they can be solved individually, via linear equation-system solvers. In a segregated solver, the momentum equations are solved first – taking other momentum quantities, turbulent quantities, pressure values, etc., from the previous non-linear iteration. Rather than solve the continuity equation, the equation is re-arranged as a *pressure-correction* (or just 'pressure') equation; essentially computing offsets required to the just-computed velocity field in order to satisfy continuity. The velocities are then corrected as necessary, and the additional transport equations can be solved using a flow field that is guaranteed to conserve mass. A coupled solver is sometimes used as an alternative, whereby the momentum and pressure equations are solved together, in one large linear system, followed by additional transport equations.

At the start of each non-linear iteration, Neumann or Dirichlet boundary conditions (which close the system in a suitable way), may need to be updated. In advanced CFD codes, meshes may be dynamic and updated between non-linear loops too. A suitable end for the non-linear loop is found by monitoring the maximum (L_∞) or root-mean square ($L2$) residuals of each equation's linear equation-system.

In order to stabilize the iterative process, under-relaxation is applied between each non-linear loop. This damps changes in the solution to prevent rapid divergence, and cancels out when the solution is converged. Relaxation can be applied explicitly or implicitly. Explicit relaxation is applied once a solution vector has been computed, by taking a weighting of the computed value and the old value:

$$\mathbf{x}_w = \mathbf{x}\alpha_{exp} + (1-\alpha_{exp})\mathbf{x}_{w-1} \tag{2.10}$$

For implicit relaxation: during the assembly of each set of equations, the diagonal of the matrix, $\mathbf{D} = \text{diag}(\mathbf{A})$, is magnified by some factor and cancelled on the right-hand side by the same amount (but using the previous solution):

$$\left(\mathbf{A} + \frac{(1-\alpha_{imp})}{\alpha_{imp}}\mathbf{D}\right)\mathbf{x}_w = \mathbf{b} + \frac{(1-\alpha_{imp})}{\alpha_{imp}}\mathbf{D}\mathbf{x}_{w-1} \tag{2.11}$$

In an unsteady flow, the transient term, which was discretized using backwards differencing, must be iterated in an even coarser loop, with each non-linear (steady) flow solution representing a single timestep. This loop is given the counter t; and only advances when the non-linear loop reaches convergence (or after a predefined number of iterations, $w_{\max}$).

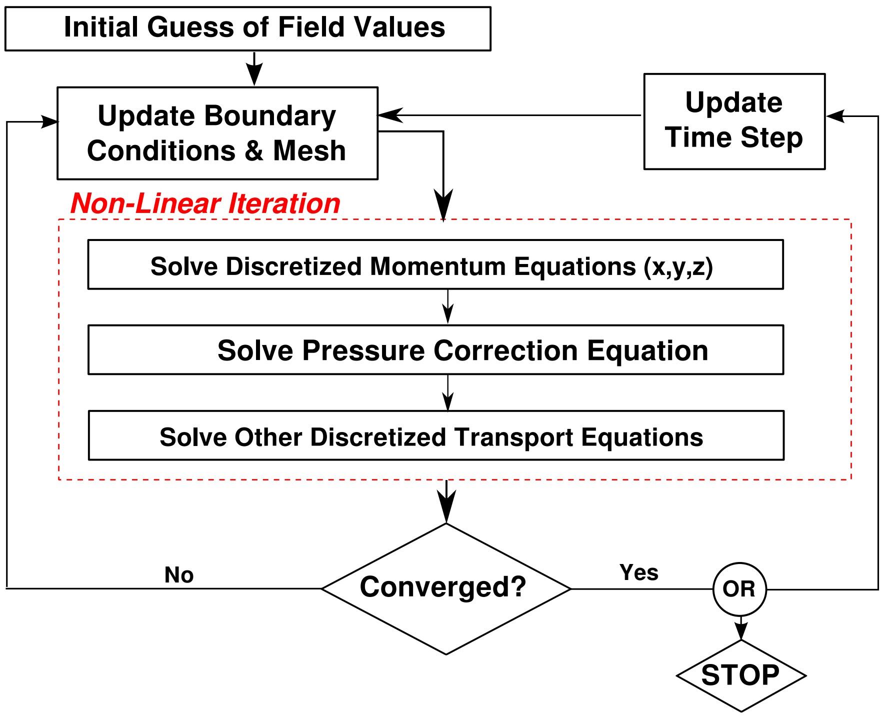

Initial Guess of Field Values
Update Boundary Conditions & Mesh
Update Time Step
Non-Linear Iteration
Solve Discretized Momentum Equations (x,y,z)
Solve Pressure Correction Equation
Solve Other Discretized Transport Equations
No
Converged?
Yes
OR
STOP

Figure 2.4: A diagram of the SIMPLE algorithm, showing the non-linear loop and the time-stepping loop.

Figure 2.4 shows the SIMPLE algorithm, and therefore the two coarsest loops of a typical CFD simulation. The routines within these loops are important to examine, from a computational perspective, since they are repeated many hundreds or thousands of times in a particular simulation. The routine is virtually the same for each equation, thus here we examine the process of solving any general linearized transport equation.

Figure 2.5 shows the process, which can be split into 4 main sub-routines.

- **Assembly**

  This routine is responsible for creating a system of N linear equations, where N is the number of cells in the discretized mesh. The equation system will have the form $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a constant N-by-N sparse matrix, $\mathbf{x}$ is the solution vector and $\mathbf{b}$ is the constant right-hand-side vector (both of length N). The process involves applying all discretization, interpolation, eccentricity corrections and orthogonality corrections. All higher-order discretization schemes apply their

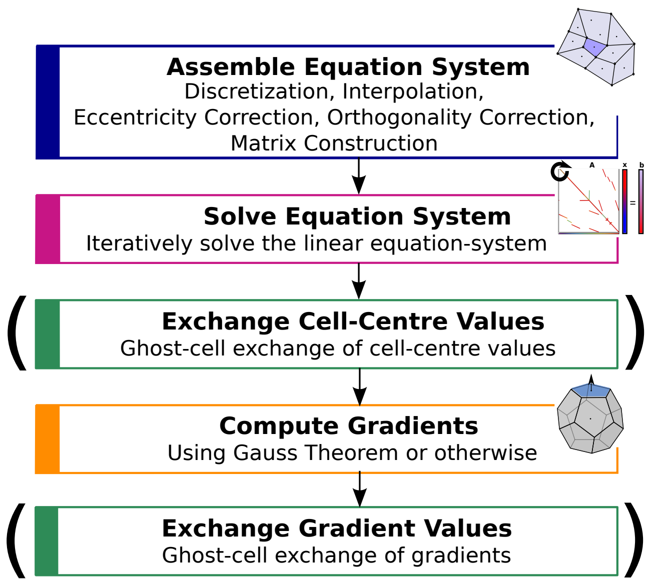

Assemble Equation System
Discretization, Interpolation,
Eccentricity Correction, Orthogonality Correction,
Matrix Construction
Solve Equation System
Iteratively solve the linear equation-system
Exchange Cell-Centre Values
Ghost-cell exchange of cell-centre values
Compute Gradients
Using Gauss Theorem or otherwise
Exchange Gradient Values
Ghost-cell exchange of gradients

Figure 2.5: A diagram illustrating the steps required to solve an individual, linearized transport equation.

low-order terms, implicitly, to the left side of the equation system; and their higher-order terms explicitly to the right side of the system, using values from $w - 1$, in a process known as deferred correction. The sparse matrix will represent the connectivity of the underlying mesh. Figure 2.6 shows an example of a sparse matrix, demonstrating the simple connectivity pattern of a two-dimensional Cartesian mesh.

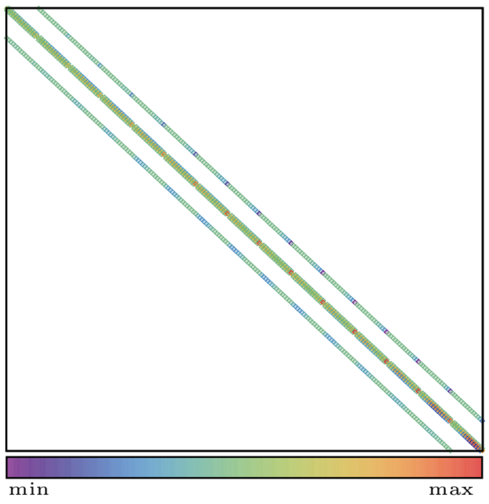


Figure 2.6: An example of a sparse matrix constructed for a simple two-dimensional, square domain.

- **Solve**

The linear equation system must then be solved. Due to the size of N, direct solvers which completely invert $\mathbf{A}$ to find the solution vector are impractical. Iterative solvers are used to rapidly approximate the solution to each linearized equation system, and this iterative loop is called the *inner loop*, with counter k, thus creating a third nested loop in the overall CFD algorithm. The residual of this system, $\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}$, can be monitored until a desired *inner-loop convergence tolerance* (ILCT) is reached. There are many iterative solvers to choose from, often with many deeper subroutines, and a more thorough examination will be performed later in this thesis (chapter 5).

- **Gradients**

  Using Gauss theorem, the gradients of the transported variable must be calculated, so that they are available for other equations. Any necessary orthogonality and eccentricity corrections must be applied.

- **Exchange**

  When computing in parallel, the discretized mesh may be partitioned across computing nodes which do not have direct memory-access to the variables after they have been computed. In this case, data exchange must occur along the boundaries of the partitions, via *halo* or *ghost* cells, such that each partition has enough information to perform its discretization and interpolation. This data exchange must happen after the initial linear solution (so that ghost data is available to compute gradients), and also to communicate the gradients once they are computed. Data exchange also occurs as part of the linear solution process, but this can be considered part of the solve routines.

As will be demonstrated later, the majority of computational effort is spent in the solve routines when using parallel computing. Most of the other named routines, above, are efficient (including assembly, calculation of gradients, time-stepping, updating boundary conditions); although every case is different. Short simulations may be dominated by one-time overheads such as file input/output; or cases involving large amounts of dynamic meshing may spend more time in those relevant routines.

2.1.3 ReFRESCO

The work presented in this thesis focuses on the development of *ReFRESCO* – a typical viscous-flow CFD code. ReFRESCO solves multiphase, unsteady, incompressible flows with the Reynolds-Averaged Navier Stokes (RANS) equations, complemented with turbulence models, cavitation models and volume-fraction transport equations for different fluid phases [99]. ReFRESCO represents a general-purpose, finite-volume CFD code, with state-of-the-art features such as moving, sliding and deforming grids and automatic grid refinement – but has been verified, validated and optimized for numerous maritime industry problems. ReFRESCO is currently being developed at *MARIN* (Netherlands), who were the sponsor of this PhD, and also part of the supervising team. Fifteen months of this PhD were spent working in the MARIN offices in the Netherlands, directly alongside the main developers. The development of ReFRESCO is highly collaborative, with contributions from IST (Portugal) [76], the University of Sao Paulo (Brazil) [80], the Technical University of Delft (the Netherlands) [52], the University of Groningen (the Netherlands) [7] and now from the University of Southampton (UK)

[36, 37, 38, 39, 40]. The University of Southampton/MARIN collaboration began in 2012, during an internship, which focused on turbulence and transition model verification and validation, along with the University of Sao Paulo [81].

The overarching theories and processes in ReFRESCO have been described broadly in the previous sections. To summarize, ReFRESCO is a finite-volume, viscous-flow CFD code, capable of modeling RANS, LES, and hybrid-RANS/LES. The code is based on the SIMPLE algorithm, with its three nested loops (t,w,k) for unsteady time-stepping, non-linearity/equation-coupling and the iterative solution of the linearized equations. ReFRESCO is 'face-based' with cell-centred collocated variables, designed to handle unstructured grids in a pseudo-object-oriented manner. Nothing prevents ReFRESCO from using structured or block-structured grids (which are often of higher quality than unstructured meshes), but this scheme does not take advantage of the inherent structured data/memory layout of these grids (unlike pure structured-grid CFD solvers).

ReFRESCO is written in free-format Fortran, and linked to many state-of-the-art external tools. ReFRESCO uses MPI (Message Passing Interface), specifically OpenMPI and IntelMPI, to perform communications between parallel partitions – including data exchange routines and communications occurring inside the solve routines [66]. ReFRESCO uses a single-level parallelization scheme (MPI-only), and does not use other layers such as OpenMP threading which will be described in the following section.

METIS [27], a tool for accurate and fast partitioning of unstructured meshes, is used to partition the domain into parallel chunks. METIS attempts to make a fair trade-off between computational work-load (number of cells per MPI domain) and communicational work-load (size of domain boundaries or number of 'halo' cells).

ReFRESCO relies upon PETSc (Portable Extensible Toolkit for Scientific Computing), for most of its linear algebra needs. The work of the solve routines is completely off-loaded to PETSc, by building the linear system ($\mathbf{Ax} = \mathbf{b}$) inside PETSc, and using PETSc's suites of linear solvers and preconditioners [6]. PETSc is written in C, but has a Fortran interface which is used by ReFRESCO. PETSc has an object-oriented structure, with objects for matrices, vectors, preconditioners and 'KSP's (Krylov subspace solvers). During this PhD, ReFRESCO has also been linked to Trilinos, as a direct substitution for PETSc [41]. Trilinos is similar to PETSc, but provides alternative parallelization schemes (including GPU/co-processor acceleration) and is written in heavily-templated C++. Trilinos is a collection of packages including Ifpack2 (preconditioners), Belos (solvers), Kokkos (parallelization schemes) and Tpetra (matrices/vectors). The interface to Trilinos is similar to PETSC, but is in C++, so it was linked to ReFRESCO using ISO-C bindings and a wrapper. Unfortunately, Trilinos underperforms (by an order of magnitude) compared to expectations and the cause of this has not been identified.

Although ReFRESCO is a maritime-focused CFD code, most of the fundamental algorithm is unaffected by this specialization. There are differences in the transport equations (free-surface, cavitation, turbulence, etc.), the additional features (sliding meshes, for propellers, for example) and the validation test cases – but overall, Re-

FRESCO is a general-purpose semi-implicit CFD solver. It is used for a wide variety of applications, including in the design of maritime vessels, offshore structures (especially for the oil & gas industry) and renewable energy industries (tidal and wind energy).

2.2 High-Performance Computing (HPC)

For a detailed understanding of high-performance computing, the author recommends Kogge et al. [54], which offers a very in-depth view of modern supercomputers. Saltzer & Kaashoek [84] helps to give a complete picture, by explaining the design and operation of computers in general.

A supercomputer (*a.k.a.* a *cluster* or a *high-performance computer*) is a large computer, consisting of many small *nodes* which resemble individual computers. The nodes are connected using a high-speed *inter-nodal* network, for fast data transfer; and a slower Ethernet connection for management functions.

The inter-nodal network is often copper-wire based, although optical interconnects are becoming popular for their higher speed and power efficiency [17]. *Iridis4*, the University of Southampton supercomputer, uses a copper-wire *Infiniband* network, grouping ∼30 nodes together with a 14 Gbit/s connection, and connecting these groups via four (parallel) 10Gbit/s connections in a tree-like hierarchy. Other networks, often designed for higher performance, include multi-dimensional torus connection patterns.

Nodes are the coarsest level of parallelization in a supercomputer. There is no shared memory between independent nodes, so data must be shared by sending messages (packets of data) across the inter-nodal network. Programming standards such as MPI (Message Passing Interface) are commonly used to parallelize scientific code at this level. For example, MPI communications are used in ReFRESCO to handle ghost-cell data exchange. In this programming model, each node runs a separate instance (*process*), or several instances, of a parallel application, each with its own memory, sharing data via MPI and memory buffers. This level of parallelization is often referred to as Distributed-Memory Parallelization.

An individual node is normally a fully-functional computer in its own right, consisting of central processing unit(s) (CPU(s) or *processors*), random-access memory (RAM or just *memory*), and specialist controllers which interface with hard drives, external networks (such as Ethernet or Infiniband), and much more. These devices are connected via the front-side bus (FSB). Figure 2.7 shows the layout of a typical node, with two CPUs on a shared FSB.

Often, multiple processors are placed on a single node in *sockets*, each with their own set of memory which can be directly accessed via an integrated memory controller (IMC) on each CPU. Processors can access their own memory quickly via the IMC, but they can also access the memory of other sockets via the FSB. This composition is known as symmetric multi-processing (SMP). Iridis4 consists of two symmetric sockets per node.

An individual CPU consists of various, complex sub-systems which perform particular tasks, but at its heart are processing *cores* which are fed pipelines of data to operate on

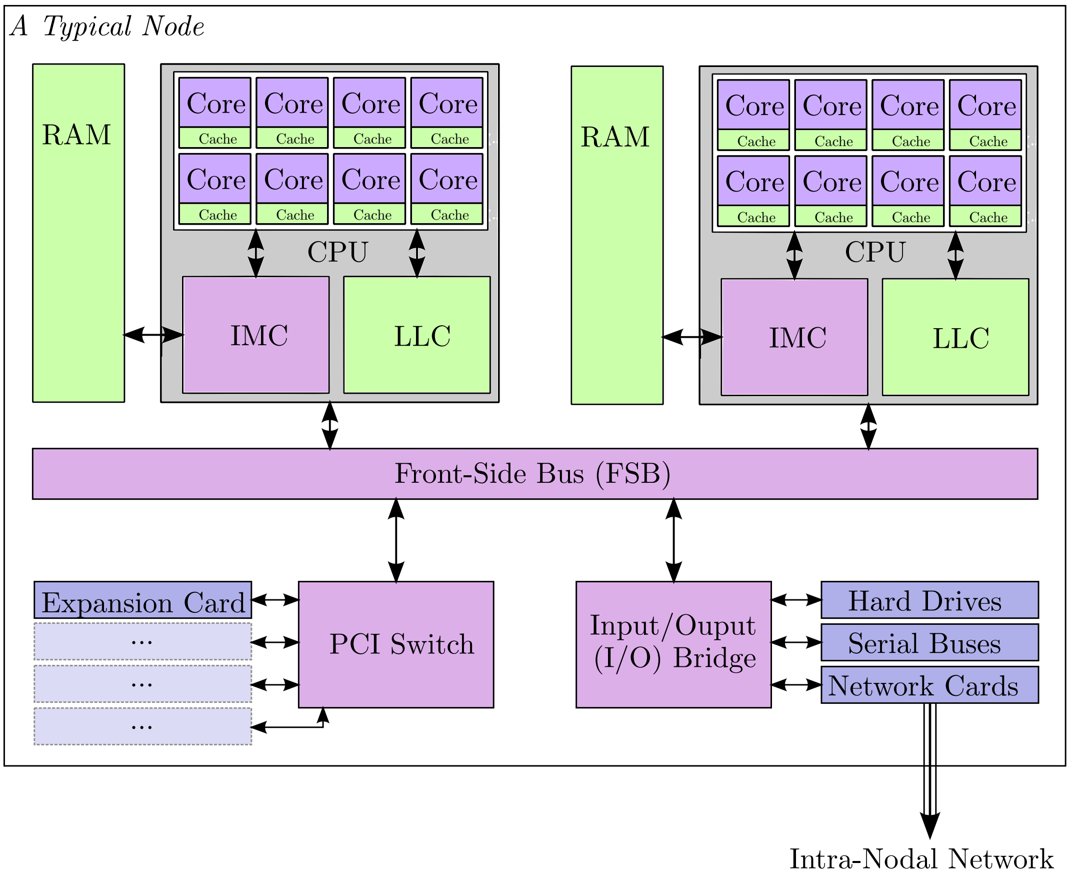


Figure 2.7: A diagram of a typical node, consisting of two 8-core CPUs on a shared FSB. Brands such as AMD and Intel have variations on the standard FSB design, such as Quick-Path Interconnect (QPI).

and instructions which must be performed. As well as the IMC, CPUs often contain small amounts of on-board memory, called *cache*, which can be used to store small amounts of re-usable data or to pre-fetch data from RAM. In the case of Iridis4, which comprises of Intel Xeon E5-2670 CPUs, there are 8 cores per CPU, each with a level-1 (L1) instruction cache of 32KB, an L1 data cache of 32KB and a 256KB L2 cache. There is also a shared 20MB L3 LLC (last-level cache) which every core can access. The higher levels are larger and slower, as they move further from the physical cores. This process of combining multiple cores on a single processor is called *chip-level multiprocessing* (CMP).

In the eyes of the programmer, SMP and CMP can be considered the second level of parallelization. Memory is shared between all cores on a node, so message-passing is no longer necessary in order to perform parallel computations. A common standard for enabling parallelization at this level is OpenMP (Open Multi-Processing)[75]; which creates *threads* (often one per core) operating on different segments of the available data within a single process. It is possible to take advantage of both inter-nodal and intra-nodal parallelization by performing MPI communications between processes, on separate nodes, and using multi-threading within each node – this is known as *hybrid parallelization* or *MPI+X*. It is also common to run multiple processes on a single node, each with their own memory partition, thereby using MPI for intra-nodal and inter-nodal parallelization. These *MPI-only* implementations may struggle to compete with multi-threading at the intra-nodal level, due to unnecessary memory copying and inability to effectively use shared cache.

An important concept at the intra-nodal level is that of *non-uniform memory access* (NUMA). When data is accessed by a core, it could come from L1 cache, L2 cache, L3 cache, the local memory (via IMC) or SMP memory (via FSB); and although compilers optimize data locality as much as possible, asynchronicity naturally occurs. It is often the job of the programmer, or algorithm, to hide this asynchronicity or to improve data locality.

In a modern computer, so-called *accelerators* are often added to nodes as expansion cards. These expansion cards could be graphics processing units (GPUs) or specially-designed coprocessors, and they communicate via a PCIe (Peripheral Component Interconnect Express) switch which connects to the system bus. These accelerators act in a similar way to a node, with their own memory and processor (typically with many more cores than a CPU). Communication and scheduling of parallel workloads is handled by the host node, offloading data and instructions via standards such as CUDA [71], OpenCL [92], or (more-recently) cross-vendor OpenMP [75] and OpenACC [73] standards. Newer co-processors, such as Intel's Knight's Landing, have network cards which allow them to act as nodes in their own right, communicating via MPI.

Another level of paralellization occurs at the sub-core level, sometimes referred to as vectorization, single-instruction-multiple-data (SIMD) parallelization, superscaling or data-level parallelization (DLP). If there are blocks of aligned memory (*i.e.* vectors) which are operated on in a non-dependent loop; it is possible to perform operations on much of the data at the same time. The size of the SIMD register determines the size of vectorized blocks – for example, a 256-bit register will allow blocks of 4 double-precision floating points numbers to be operated together. In ideal situations, this is a 4x speed-up. For processors based on the x86 architecture, vector processing is handled by the Advanced Vector Extensions (AVX) protocol – which constantly improves by offering larger SIMD registers and improving the amount of operations that can be vectorized.

Compilers can do a reasonable job of auto-vectorization; but they must be conservative when evaluating data-dependency. As such, it is often the job of the programmer to exploit SIMD parallelization. Unfortunately, for applications without well-aligned memory (such as unstructured-grid CFD codes) it becomes difficult to use vectorization effectively due to limited memory bandwidth.

The final level of parallelization, referred to as pipelining or instruction-level parallelization (ILP) is handled automatically by the compiler. It allows, for example, pre-fetching data whilst computing by overlapping different operations, performed by different processing elements. Pipelining improvements are a means of improving core efficiency, allowing more FLOPS without increasing core clock rates. Figure 2.8 shows an illustration of pipelining.

This section has given an overview of a classical supercomputer architecture, following

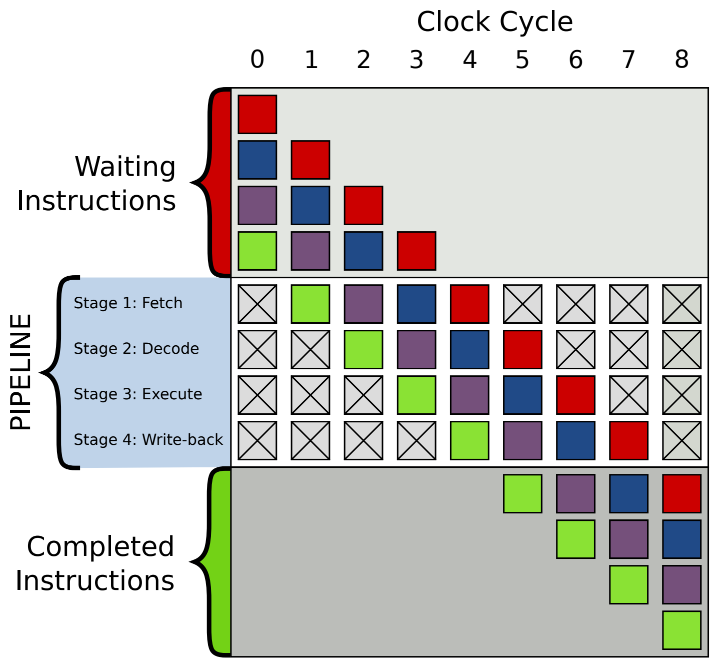


Figure 2.8: An illustration of pipelining/instruction-level parallelization (ILP). Colours represent operations which can be performed concurrently *(Creative Commons: Colin M.L. Burnett)*.

the hierarchy of parallelization schemes from the coarsest to finest level:

1. Distributed-memory parallelization, at the inter-nodal level, handled via a message-passing interface.

2. Shared-memory parallelization, at the intra-nodal level, handled via shared-memory threading interfaces or by a distributed-memory-style message-passing interface.

3. SIMD vectorization, at the sub-core level. Handled via compiler directives or subsets of the OpenMP standard.

4. Pipelining or instruction-level parallelization (ILP), at the sub-core level, automatically handled by the compiler.

Accelerators or co-processors, tied to a host but with their own distributed memory, can be considered an additional layer of parallelization – handled via specialist standards such as CUDA, OpenCL, OpenACC, or subsets of the OpenMP standard.

Over the years the emphasis on different levels of paralellization has changed; with a trend towards finer levels of parallelization. It is critical that algorithms used within scientific applications, including CFD, are designed to co-exist within this parallel framework, and are capable of utilizing multiple layers of parallelization. In the literature review, the historical trends and future trajectories of high-performance computing are demonstrated; showing where developers of CFD should be focusing.

2.2.1 Iridis4

Iridis4, the University of Southampton supercomputer which has been used throughout this PhD, has been introduced as an example above. Iridis4 has 750 nodes, consisting of two Intel Xeon E5-2670 Sandybridge processors (8 cores, 2.6 Ghz), for a total of 12,200 cores. Each 16-core node is diskless, but is connected to a parallel file system, and has 64GB of memory split into two NUMA domains (one per processor). The nodes run Red Hat Enterprise Linux version 6.3. Nodes are grouped into sets of 30, which communicate via 14 Gbit/s Infiniband. Each of these groups is connected to a leaf switch, and inter-switch communication is then via four 10 Gbit/s Infiniband connections to each of the core switches. Management functions are controlled via an ethernet network.

Iridis4 ranked #179 on the Top500 list of November 2013 with a peak performance of 227 TFLOPS [96]. Iridis4 cannot be classified as a next-generation, many-core machine, with only 16 cores per node. Indeed, it is several years behind the state-of-the-art. Nonetheless, it should be able to give sensible insight into the limitations of the CFD algorithm if appropriately-sized simulations are used.

2.3 Closure

This chapter has given an overview of the fundamentals of CFD and HPC, and described how they cooperate. It has also introduced the CFD code used during this thesis (ReFRESCO) and the supercomputer on which it is run (Iridis4).

The decomposition of the CFD methodology into the key routines of the SIMPLE algorithm, {assembly, solve, gradients, data exchange}, will be used later to break down the performance of ReFRESCO, and identify areas where scalability is poorest. Meanwhile, the knowledge of HPC-related technologies, such as cache, vectorization and MPI communications, will help identify the cause of poor scaling and guide the development of algorithmic improvements.

The following chapter will build on this background chapter, by considering the state-of-the-art of CFD and HPC, and how both these environments may be changing in the near future.

Chapter 3

Literature Review

Literature relating to the future of HPC and CFD will be examined herein. To begin, the state of next-generation 'exascale' computing is examined, providing the main motivation for the research question of this thesis and defining the 'strong scalability' problem which CFD must overcome (§ 3.1). In order to answer the research question, and tackle the strong scalability problem, there are two complementary fields of research which will be investigated. Broadly-speaking, these fields are inter-disciplinary – taking aspects of computer science and applied mathematics, respectively.

Firstly, the methods of parallelizing the CFD algorithm may need to be adapted to suit modern architectures. This may include, for example, moving the existing implementations to accelerated devices or hybrid-parallel schemes which utilize shared-memory more efficiently. Parallel-programming paradigms for CFD will be explored in section 3.2.

Secondly, fundamental changes to the core CFD algorithms may be required in order to improve strong scalability. These algorithmic changes are not necessarily dependent on any particular architecture or parallel-programming model; although it is difficult to avoid cross-over between these two fields. Areas in which numerical algorithms are being improved for strong scalability and performance are discussed in section 3.3.

Briefly, other developments in the overall CFD methodology are also discussed, in order to understand how developments towards exascale CFD must merge with other areas of research.

More specific literature will be introduced in subsequent chapters where it is more relevant.

3.1 Exascale Computing & The Strong Scalability Problem

The Top500 list, which has recorded the development of state-of-the-art supercomputers since 1993, provides excellent insight into historical trends of hardware capability [96]. The maximum performance (based on Linpack, a linear algebra solver and benchmark tool) has doubled approximately every 13.85 months, considerably faster than Moore's law would predict (18-24 months) [28] – see figure 3.1.

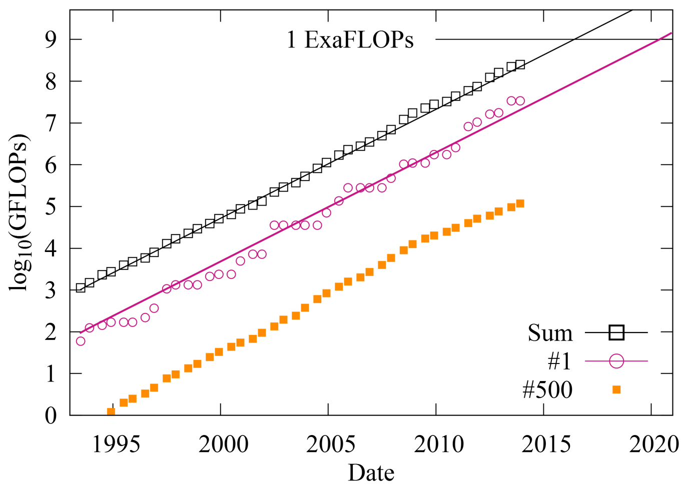


Figure 3.1: Growth of the Top500 supercomputers since 1993, showing exponential growth in performance of the #1 machine, the # 500 machine, and the sum of the Top500 machines. Data compiled from [96].

The first exascale computer, capable of one exaFLOPs (10^{18} floating-point operations per second), should be in operation by 2020 according to this trend. This is a 10.75-times performance increase compared to the current #1 supercomputer (Sunway TaihuLight). The challenges associated with reaching exascale will redefine the entire high-performance computing ecosystem, and fundamentally break the current programming paradigm. Some make the argument that the first exascale machine will be late, due to the complexities that must be overcome [46], and this seems quite likely. A new supercomputer, *Aurora*, commissioned by the Argonne Leadership Computing Facility, is scheduled for 2018 with a peak performance of 180 PFLOPS – which is slightly behind current growth trends.

In 2008, a report was commissioned by DARPA (Defense Advanced Research Projects Agency) which, in part, assessed the possibility of exascale computing [54]. Despite being written in 2008, most of the predictions have held true so far, and it is still used today as a valid analysis of exascale computing. There are a few key points to extract from this 297-page report:

- The amount of energy required to move one bit of data along a copper wire remains constant, whilst the energy cost of floating point operations decreases. As the FLOP rate of HPC systems increases, the power requirements of inter-nodal communication becomes a driving factor. Using the current growth trends, an exascale computer would consume over 60MW of electricity (around $60 million US dollars per year); to bring this value to manageable levels (around 20MW) the rate of increase of inter-nodal communication must decrease. There are advances in optical interconnect technologies, which aim to improve bandwidth and efficiency compared to copper-wire interconnects [17]; but their development cannot keep up with current node-scaling rates. Essentially, node-scaling will slow down and computational power must come from elsewhere.

- Core clock rates have stagnated, since 2004, due to a breakdown in Dennard's Law [23]. Dennard's Law states that as transistors become smaller, their speed increases linearly and their total electrical power consumption decreases quadratically. Quantum effects, causing electrons to 'leak' across the silicon insulators, cause leakage currents in modern transistors – thus breaking the latter part of Dennard's Law. To avoid runaway power consumption due to leakage currents, core clock rates must remain constant. Indeed, core clock rates may even decrease to allow a larger margin for other electricity-consuming systems such as inter-nodal communications.

  Fortunately, some gains can still be made without increasing power consumption. Pipelining and other similar technologies allow instructions-per-cycle to increase, allowing a greater FLOP rate without increasing clock frequencies. Despite this, to achieve an exponential growth rate in total performance, whilst limiting the number of nodes and decreasing core clock-rates, a huge growth of intra-nodal (cores-per-node) concurrency must occur.

  A modest increase in intra-nodal parallelization is already apparent. Since 2004, manufacturers have been packaging multiple cores into single processors in order to keep up with exponential performance increases. Figure 3.2 shows how the performance of the Top500 supercomputers can be attributed to core-clock speeds and number-of-cores since this shift towards a multi-core architecture.

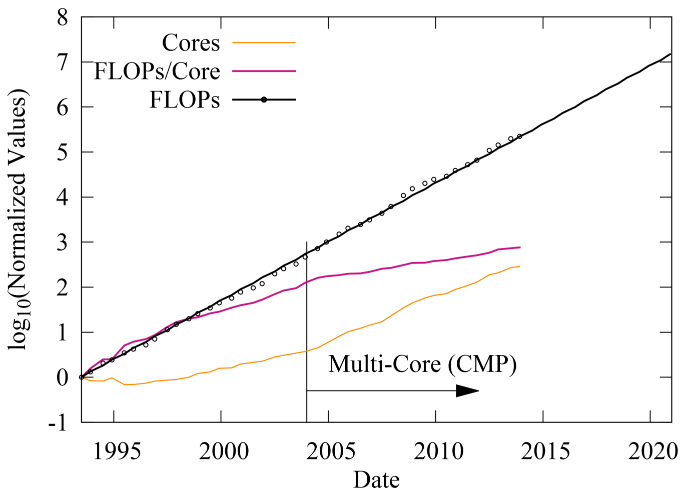


Figure 3.2: Growth of the Top500 supercomputers since 1993, showing how total performance is attributed to total parallelization and the FLOP rate of individual cores. Note the critical point in 2004, where FLOPS-per-core began decelerating and parallelization began accelerating. Data compiled from [96].

An exascale machine is likely to have O(1k) to O(100k) cores per node, depending on the specific architecture. Similarly, total concurrency is likely to grow to an order of 1 billion cores (x10-100 for latency hiding). Sunway TaihuLight has only 10.6 million cores, so this is a growth in parallelization of approximately 100$\times$ in just three years. This corresponds to over a 25$\times$ acceleration of parallelization rates, compared to the trends illustrated in figure 3.2. This sudden acceleration demarks the end of the multi-core era, and the start of the 'many-core' era.

- Memory bandwidth grows more slowly than processor-performance (approximately half the speed), so data locality, cache usage and latency-hiding will become more important. Memory capacity is also growing at a similar rate, which limits the maximum size of scientific simulations [46].
- Heterogeneous architectures are likely to become the norm. Most likely this will mean that nodes consist of multiple types of core, each specialized for different tasks. This is already visible in accelerated nodes, where CPU cores are augmented by vectorized GPU cores or low-speed co-processor cores. This system may be adopted within individual CPUs; where various types of core exist to achieve specialist tasks more efficiently. This trend is partly driven by the consumer market for embedded systems: tablets, phones and laptops already use heterogeneous processors (such as for embedded graphics).

  Some aspects of heterogeneity are already visible on regular CPUs. Modern CPUs down-clock cores when operating near the thermal design envelope in order to maintain efficiency without sacrificing peak performance; thus creating a pseudo-heterogeneous environment. This is often marketed from the opposite perspective, as a 'turbo' feature. This concept, known as 'dark silicon' will become commonplace and much more pronounced. In Esmaeilzadeh et al. [25], the authors discuss the power challenges facing exascale computing, and predict that the homogeneous-core architecture will not be sufficient in achieving exascale, due to the sheer number of cores which must 'go dark'. Indeed, the authors propose that specialized, heterogeneous cores are needed.
- Due to increasing pressure to reduce operating voltages the MTBF (mean time between failure) of individual cores will decrease. Combining this effect with the sheer number of cores participating in a simulation, it will become necessary for simulations to survive core-failures. It is not clear whether this will come entirely from low-level systems (such as the operating system or parallel-programming libraries) or whether user codes will each have to adapt.

Many of these trends are already visible with the recent growth of many-core supercomputers, which exhibit higher FLOP/MW ratios in response to energy efficiency constraints [46]. *Tianhe-2* (the #2 fastest supercomputer in the world, according to the Top500 list [96]) uses 3 Xeon Phi co-processors to accelerate each node, providing 192 cores-per-node (totalling 34 PFLOPS at 17.8MW). The #3 supercomputer, achieving 17.6 PFLOPS (at 8.2MW), uses k20 graphics cards providing 2496 low-speed cores-per-node. However, it could be that accelerators are just a stepping-stone to many-core CPU-based architectures.

Sunway TaihuLight (#1 of the Top500 [96]) uses 260-core CPUs to achieve a peak performance of 94 PFLOPS with a power consumption of 15.3MW. Each processor contains 256 vectorizable, low-frequency cores and 4 more-conventional cores. It is a heterogeneous architecture with just 123MB of memory per core (32GB per node). To put this in perspective, *Iridis4* contains 64GB of memory per node, but only 16 cores.

These new machines, and upcoming supercomputers such as *Aurora* [4], continue to validate the predictions made in Kogge et al. [54]. Changes in architecture will require the fundamental parallel programming paradigms of CFD to adapt. Furthermore, huge growth in parallelization will require the CFD algorithm to scale more efficiently.

In the following two sections, the way in which CFD has, or can, adapt to this next-generation computing environment will be examined. Firstly, programming paradigms will be considered, looking at ways in which CFD can adapt to make use of new hardware. This will focus on computational changes to the code – using different parallelization techniques, exploiting accelerators, and examining data-level (SIMD) parallelization – without changing the numerics of the code. Secondly, the numerical algorithms which form the core of the CFD methodology will be examined, identifying areas where the numerical algorithms themselves have already been adapted for strong scalability, and where further work is required.

3.2 Parallel Programming Paradigms for CFD at Exascale

Most commercial and open-source CFD codes are based on an MPI-only model, including ReFRESCO [99], Star-CCM+ [13, 14, 15], OpenFOAM [74, 77] and ANSYS CFX [19] – in general, their methodology, performance and scalability are similar. However, there is now an increasing amount of literature discussing alternatives to this MPI-only model. For example, there have been experimentations with hybrid parallelization in OpenFOAM [20] and GPU-acceleration in ANSYS CFX [42]. In this section, this literature will be investigated in order to understand the state-of-the-art with respect to CFD and parallel programming paradigms – including hybrid parallelization, accelerators and vectorization.

3.2.1 Hybrid Parallelization

A popular concept for strong scalability is that of hybrid parallelization, whereby distributed-memory processes communicate via MPI (across node or NUMA boundaries), and within each shared-memory process multiple threads operate together, using OpenMP or similar [78]. Neither ANSYS nor Star-CCM+ have mentioned hybrid programming paradigms, but they may be investigating this. Developers of OpenFOAM have experimented with hybrid paralellization with good results [20]. The hybrid schemes were shown to be faster at high core counts ($C = 4096$), but their implementation was badly optimized for memory locality, and many start-up routines were not hybridized. On routines which were well-optimized, hybrid execution was up to 15 times faster; but this speed-up was probably due to cache effects or better optimization and should not be considered a benchmark.

This OpenFOAM study shows some benefits of hybrid paralellization with respect to strong scalability and large core-counts; further improvements to memory locality and more wide-spread hybridization may make this an accessible performance gain for CFD. PETSc, the scientific algebra library on which ReFRESCO depends, has been fully hybridized with modest results [56], with some performance increase when task-based threading was used (dedicating one thread purely to MPI communications). However, official support for the hybrid parallelization model in PETSc was recently dropped. Instead, the PETSc developers hope that MPI-only will remain relevant as MPI standards develop over the next decade. There have been many interesting debates on their mailing lists regarding this decision.

Some of the advantages of hybrid parallelization include better memory capacity usage (reducing duplication of data within the same node), better memory locality, better usage of inter-nodal networks and dynamic fine-grained load-balancing [78, 90]. That being said, numerical codes using non-hybrid schemes can also realize these advantages, albeit with greater programming effort.

3.2.2 Accelerators

Accelerators include graphics-processing units (GPUs) or co-processors (such as Intel's Xeon Phi) which are attached to a host node. Usually, these devices are connected via the PCI serial interface to the host node (see figure 2.7). The Xeon Phi can be operated as an MPI node in its own right, or work can be off-loaded (either partially or entirely) by OpenMP directives. GPUs work in a similar way to off-load-mode Xeon Phis, using programming standards such as CUDA or OpenCL. Functions performed on GPUs (*kernels*) must be re-written in an appropriate syntax; whereas Xeon Phi's understand the same instructions as the host (thus they are much easier to use). All forms of accelerators are designed to provide large throughput computing power; favouring SIMD (Single-Instruction-Multiple-Data) vectorized routines. This is opposed to CPUs which are latency-optimized, focused on performing complex, varied tasks via large caches and diverse processing elements. The usage of accelerators is usually a trade-off between the speed-up obtained from using high-throughput processing compared to the cost of moving data to the accelerator.

Gorobets et al. [32] presents one of the most comprehensive CFD-acceleration studies, translating the vast majority of CFD operations into OpenCL kernels – including those for discretization, interpolation, gradient computation and linear-equation-system solving. The author demonstrates that the routines run 7.6 – 22.5x faster on an NVidia C2050 GPU, and 11.6 – 96x faster on an AMD 7970 GPU, as compared to the run-time on a standard Intel CPU (X5670). However, comparisons between CPUs and accelerators are not very revealing (their hardware is fundamentally different). The author goes on to show that the GPU kernels achieve very poor throughput compared to their theoretical peak; and that the CPUs are used more efficiently. This is mostly due to the limited memory bandwidth and host-to-accelerator communication which creates a bottleneck. Furthermore, the code used by Gorobets et al. used a structured (Cartesian) mesh which could be vectorized for efficient computation on GPUs.

Soukov et al. [88] performed a similar study, specifically looking at higher-order discretization routines for unstructured CFD meshes accelerated on GPUs. The GPU shows good performance up until memory bandwidth is saturated. In order to compare between CPUs and GPUs, the author shows that the speed-up is greater than the relative electrical power consumption of the two devices (the GPU requiring 2.5x more power, but producing a larger speed-up). Similarly, Rossi et al. [82] presents GPU acceleration of an unstructured finite-element code. The accelerator becomes more efficient as the test case becomes larger – when 1-million cells were offloaded the GPU became up to 4.11x faster than the CPU.

Gorobets et al. [33] performed a study comparing the performance of NVidia GPUs, AMD GPUs and first-generation 5110P Xeon Phis for unstructured discretization routines. The Xeon Phi performed poorly compared to the GPUs, but the author provides many explanations for this. Most notably, the code is optimized for massive multi-threading; and this is only possible in embarrassingly-parallel parts of the CFD code (such as in the assembly and gradients routines). The Xeon Phi behaves more like a standard CPU, with larger caches and more hierarchical memory access, which should allow it to maintain speed-up on more complex routines (with irregular memory access or inter-thread communication). In Liu et al. [59], an NVidia K20X GPU is compared to a Xeon Phi for sparse-matrix-vector-multiplication (SpMV) routines, which feature heavily in the solve routines for CFD. Xeon Phi's were considerably faster in this test case, which could be much more meaningful for overall CFD speed, as the solve routines are expected to be a bottleneck to strong scalability. Gorobets et al. notes that despite the poor comparison to GPUs in the naïvely parallel code, the Xeon Phi achieved very good internal scaling, achieving approximately 83% parallel efficiency over its 240 threads (compared to serial run-time on a single Xeon Phi thread).

The latest generation of Xeon Phi, dubbed *Knight's Landing* (KNL), was released in 2015, featuring 3x the raw compute power of the first-generation Xeon Phi [94, 95]. KNL features an even more hierarchical memory architecture, with a bank of 'near memory' (8 or 16GB) acting as a form of large cache. This is very attractive for CFD, which is often memory-bandwidth limited.

Recent developments in the OpenMP standard (4.0) support offloading of data to an accelerator without re-writing special vendor-specific kernels. The much younger OpenACC standard was created for the same purpose. Unfortunately, few compilers support the full OpenMP or OpenACC standard; with most offering partial coverage for their preferred hardware. For example, the Intel compilers support OpenMP 4.0 offloading, but only to Intel Xeon Phis. Similarly, the Portland Group compilers support OpenACC for NVidia GPUs – but nothing else.

In Horst [46], a certain amount of scepticism around the long-term usage of accelerators is presented. The author suggests that accelerators may be a short-term solution to achieve a pseudo-many-core architecture. The expectation being that these throughput cores will move on to the same package as the typical CPU cores. The reasoning is that most supercomputing technological advances are driven by changes in the consumer market. Mobile phones, tablets, and laptops typically use embedded systems, with specialist cores for general-purpose computing, graphics processing (SIMD) and other specific applications – all on the same chip. The recently-developed *Sunway TaihuLight* fits this prediction very well. In some ways, the heterogeneous, embedded vectorization cores act similarly to an accelerator, so programming standards such as OpenMP are likely to adapt well to this environment.

3.2.3 Vectorization

A key part of modern processors, including graphics cards and co-processors, are their SIMD vector extensions to the standard (x86) instruction set. As more complex instructions become vectorizable and the SIMD registers on these devices get larger, vectorization cannot be ignored [91].

Ierotheou et al. [49] performed some work on vector processing for CFD in 1989 (when array processors were popularized), showing substantial speed-up in some areas of the code. The author states that vectorization could be exploited in the solve routines, offering up to 20x speed-up for simple iterative solvers (such as Jacobi methods). The limiting factor for other routines, such as assembly routines was difficult branch prediction (conditional *if* statements inside a loop make it difficult to vectorize) due to boundary conditions and irregular source terms from the turbulence equations. Vector extensions in modern processors are quite different. Firstly, the maximum speed-up is proportional to the vector register size – on *Iridis4*, a 256-bit (4 double precision floats) vector register grants an ideal 4x speed-up. Secondly, the vectorization has become more robust – 'masking' of different conditional statements allows difficult branch prediction, and some indirect memory access is allowed. Compared to the hardware available to Ierotheou et al., speed-ups should be smaller but perhaps applied to more of the code.

Memory alignment is an important factor for effective vectorization. In a structured CFD code, where cells can be expressed in Cartesian (i,j,k) format, and relationships between cells can be expressed as constant offsets, memory access is very uniform (which leads to high performance in itself) and vectorization is trivial. In an unstructured code, which typically uses pointers to non-contiguous memory, memory alignment is much harder (and is the reason that CFD is often memory-bandwidth bottlenecked). Vectorization is often possible, but inefficient, since the vector processor will be starved of data. Attempting Intel's auto-vectorization on sections of the solve routines in PETSc results in a compiler warning: "*Vectorization possible but seems inefficient*" [51].

The operational intensity (the ratio of FLOPS per memory operation) is often used as a metric to assess the performance characteristics of numerical algorithms – often visualized on roofline diagrams (figure 3.3). The operational intensity of most unstructured CFD is low, making vectorization unproductive unless the memory access can also be vectorized.

Sparse matrix storage structures have been developed which can improve operational intensity, such that vectorization becomes possible. One such format is based on ELL (originally designed for solving ELLiptic boundary condition problems), which pads each row of the sparse matrix such that it is the same length, allowing some vectorization. Saad & van der Vorst [83] discusses the ELL format, as well as standard formats such

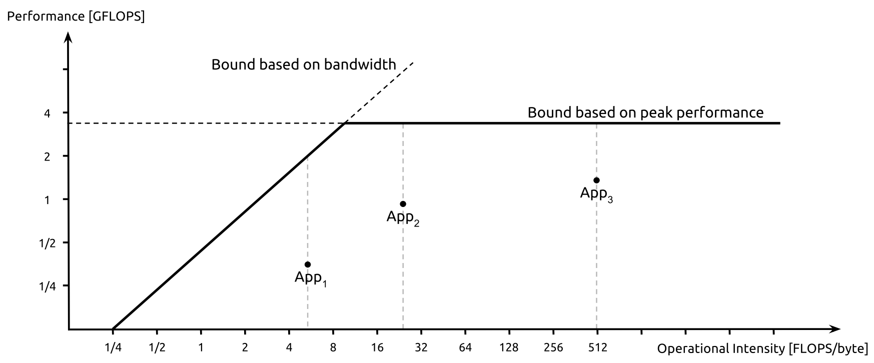


Figure 3.3: An example of a roofline diagram, used to visualize operational intensity and its affect on maximum performance of any particular algorithm [Wikimedia Commons: Giu.natale].

as CSR (Compressed Storage Rows) used by most CFD codes, including ReFRESCO. Other formats exist, such as JAD (Jagged Diagonal) with optimal vectorization, but require impractical matrix re-ordering. D'Azevedo et al. [22] shows order-of-magnitude speed-up using ELL on SpMV routines on specific Cray vector machines, which is encouraging for CFD. The status of PETSc implementation of such formats is unclear; they appear to feature in well-developed GPU kernels, but not for modern CPUs.

In Basermann et al. [9], an automatic-vectorization script called *Scout* was developed and applied to two CFD codes: one structured (TRACE) and one unstructured (TAU), using modern vector extensions. Convective discretization routines achieved perfect speed-up (x4) in the structured code, and a 2x speed-up in the unstructured code. Other assembly and gradients routines achieved up to a 50% speed-up. The ability to vectorize was heavily influenced by memory storage patterns and specifics of the implementation – which differ somewhat from ReFRESCO's implementation. The significant performance differences between structured and unstructured meshes raise the possibility of recovering structured blocks from hexahedral unstructured meshes, for the purposes of vectorization. This is done successfully in *EXN/Aero* [24].

In this section the programming paradigms surrounding exascale have been discussed, exploring the ways in which existing CFD algorithms can be adapted to perform on exascale machines. Hybrid parallelization, accelerators and SIMD vectorization have been discussed. Of the methods described, OpenMP stands out as a powerful programming standard to tackle multiple exascale challenges. OpenMP is primarily designed to handle shared-memory (multi-threaded) parallelization, which can be used to build a hybrid MPI+OpenMP parallelization scheme; but it also offers means to offload to both GPUs and Xeon Phi co-processors, and is well suited to heterogeneous CPUs. In addition, it provides the necessary directives to control SIMD vectorization. The competing schemes, such as OpenCL, CUDA or OpenACC, only address accelerators, and may be better for this specific purpose. Hybrid parallelization with MPI+OpenMP appears to be a logical starting-point for exascale CFD.

3.3 Numerical Advances in CFD

The previous section evaluated programming paradigms which may change the way numerical algorithms translate into the hardware on which they run. In this section, means of changing the numerical algorithms are examined: identifying areas where the numerics themselves could be improved for strong scalability. All the numerical algorithms in the SIMPLE scheme can be broken down into three patterns: serial computations, neighbour-to-neighbour communications and global communications; which can be used to identify their scaling patterns.

- **Serial Computations:** Any computations that are performed purely on local data. If ghost-cells are up-to-date, then all the routines in the assembly and gradients, including discretization, interpolation, corrections and matrix assembly are entirely serial computations. Serial computations scale well at the inter-nodal level, with each node providing a proportional speed-up of any serial computations. At the intra-node level serial computations may scale with number of cores or memory bandwidth – usually some complex mixture of both.
- **Neighbour-to-Neighbour Communications:** The exchange of data between ghost cells is an example of neighbour-to-neighbour communication. Neighbouring partitions of a mesh must communicate data on their borders to a handful of processes, but not every process. As the number of partitions increases, with more processes, the number of neighbours should not change dramatically. Since various neighbours can communicate simultaneously, this is (mostly) a scalable communication pattern. Eventually, a decreasing cells-per-core ratio will reduce the size of these MPI messages such that they are limited by inter-nodal latency – this is a hard limit on the scalability of these routines.
- **Global Communications:** Some routines, particularly those in the solve routines require global communication. Typically these communications are used to find global maxima, minima or averages across all cells in the domain. In the linear equation-system solvers, they are typically used to compute vector norms or residuals across partitioned vectors. Sometimes called *collective communications*, they perform a tree-like hierarchical communication pattern, where every single message is latency-bound. Global communications are not scalable, *increasing* (not even decreasing!) at a rate of $\log_2(P)$, where P is the number of participating MPI processes.

The two communication patterns can cause additional problems due to synchronization. Synchronization occurs implicitly wherever processes must communicate, in the case of two MPI processes performing neighbour-to-neighbour communications, the two processes must reach the same point in the code before they can proceed. If the two processes are not perfectly load-balanced then there is an implicit synchronization delay, as one process waits for the other. If every MPI process is involved in neighbour-to-neighbour communications at once, a *global* implicit synchronization can be created. In many situations, (*non-blocking*) communication can be overlapped with useful computations to hide communication costs, including synchronization – but as the cells-per-core ratio decreases this becomes more difficult, as the amount of serial work decreases (per process). In some cases, overlapping computations and communications is not possible at all, and a *blocking* communication is used, causing explicit synchronization. Implicit or explicit global synchronizations can cause a serious scalability problem.

The assembly and gradients routines are of relatively little interest to strong scalability; as they are entirely serial computations. Improvements at the intra-node level mostly come from computational optimization, such as vectorization, as discussed in the previous section.

The ghost-cell data exchange, which uses neighbour-to-neighbour communications, is scalable but subject to implicit synchronization delays. From an algorithmic perspective, overlapping communication and computation does improve this. Some studies have reported that data exchange routines are the main bottlneck to strong scalability of CFD [34], but it is suspected that poor load balancing reduced the quality of these findings.

The solve routines are the subject of modern scalability research. Other studies concerning CFD state that the linear equation-system solvers (or just *linear solvers*) are the main bottleneck to scalability due to their global communication patterns [18]. The pressure-correction is (by far) the hardest of the equations to solve, due to its elliptic nature [11] – as will be shown later. Modern research focuses on reducing the amount of global communication required by the solve routines [61, 108], or reverting to algorithms with worse numerical convergence, but no global communication [2, 10]. The linear solver will become one of the main focuses of this thesis, and a more complete review of literature will be presented in the relevant chapter.

There is much research into numerical methods which do not directly influence strong scalability, but may have a profound effect. It is important to be aware of these developments when attempting to make any developments to the fundamental CFD algorithms. Indeed, many of these methods put more emphasis on the linear solver, which may negatively affect scalability of the code overall.

- **Higher-Order Discretization:** With memory operations becoming more expensive relative to floating-point-operations, more complex discretization is achievable for a relatively low cost. Higher-order discretization methods such as Discontinuous Galerkin are gaining popularity for their relative cheapness and high accuracy [69]. In turn, these higher-order schemes could reduce overall solution time (by allowing smaller meshes). Discontinuous Galerkin is a powerful scheme because it does not change the discretization stencil (cells only have links to their direct neighbours), and thus the linear-equation system matrix structure remains the same. The linear system is likely to become stiffer, requiring more iterations, thus this becomes a trade-off between performing fewer outer loops but requiring more inner loops.
- **Higher-Order Linearization:** Linearization in ReFRESCO, and most other

codes, is performed by Picard's method which takes 'lagged' values from the previous outer loop ($w - 1$) to create a linear system. Higher-order systems such as variations on Newton's method, could be advantageous [43]. The trade-off is between memory requirements, additional computation steps, faster convergence, and once again – a stiffer linear equation system.

- **Pressure-Velocity Coupling:** The three momentum equations and the pressure-correction equation can be coupled together into one single linear-equation system. This requires a more powerful linear solver (with complex pre-conditioning), but can converge the overall system faster; since the burden of coupling momentum and pressure equations is transferred to the linear solver. Klaij & Vuik [52] reports reductions in the number of outer loops by a factor of 5–20, reducing computation time by factors of 2–5, depending on the particular test case. For more challenging cases, it can be very difficult to converge the coupled system, and the benefit is lost. The pre-conditioning of the coupled linear system may also be a big enough bottleneck to scalability to negate these benefits, depending on the global communication requirements. At their heart, the coupled solvers still rely on classical linear solvers, which brings the scalability concerns to the forefront once again.

  With similar motivation, the PISO algorithm (popular in OpenFOAM) computes the pressure-correction multiple times per outer loop (correcting momentum equations each time) to provide more coupling between momentum and pressure [100].
- **Parallel-in-Time Methods:** It is possible to apply parallelization to the time dimension. This is done by solving multiple time-steps at once, with a prediction for the initial conditions and a correction after each 'block' of parallel time-steps. The technology is still young, but exciting for unsteady CFD. There are issues with scalability, due to the large amount of data which must be globally communicated [24].
- **LES or Hybrid RANS-LES:** For the accurate and reliable prediction of turbulence, particularly in regions of flow separation, RANS modelling is unsatisfactory. Large-Eddy Simulation (LES) is becoming accessible due to increasing computational power; and hybrid RANS-LES models or wall-modelled LES are also becoming common [30, 107]. LES models require higher fidelity simulations, with larger grids and shorter timesteps. Since the time dimension cannot be efficiently parallelized, it is necessary to provide greater parallelization and partitioning of the spatial dimensions, in order to increase the capabilities of LES simulations. This provides additional motivation for reducing the efficient cells-per-core ratio and improving strong scalability; but the SIMPLE algorithm is unchanged.
- **Additional Physical Models:** As simulations become more complex, CFD practitioners wish to model more complex flow features. For maritime CFD, this includes transition modelling and advanced multiphase modelling (such as cavitation) [45, 80, 81]. These models typically require higher fidelity simulations

with reduced timesteps – again, motivating strong scalability of CFD.

- **Automatic Mesh Refinement:** Automatic mesh refinement/adaptation is a means to methodically increase or decrease mesh resolution according to maturing flow features in an active simulation [105]. It is an impressive technology that reduces the labour associated with creating meshes, and improves the efficiency of the CFD algorithm by adding and removing mesh resolution exactly where it is needed. From a scalability perspective, there are particular issues with dynamic load balancing, but this is a very active area of development.
- **Overset Meshing:** In a similar vein, overset-mesh technologies have been developed, which allow different parts of the domain to be meshed independently and overlapped [102]. The flow-solver dynamically cuts holes in and interpolates between the non-contiguous meshes, allowing complex geometries to be meshed easily (reducing labour) and enabling multi-body, dynamic simulations. There is a computational overhead, including difficulties in load balancing, and there are many physical and numerical challenges too. For example, the structure (or a mask) of the linear system $\mathbf{Ax} = \mathbf{b}$ will change every time the meshes move [60].
- **Immersed Boundary Methods:** As an alternative to both automatic mesh refinement and overset meshing, immersed boundary methods are an interesting area of research – completely removing the need for mesh generation and enabling extremely complex multi-body, dynamic simulations [64, 104]. Immersed boundary methods use simple meshes, such as uniform Cartesian grids, and adjust the equations of motions (rather than the mesh) to account for solid bodies in the flow. In terms of scalability, the structured-grid formulation allows better utilization of memory bandwidth and vectorization – but the overall algorithm is not significantly changed, still requiring accurate and fast solution of elliptic and non-elliptic governing equations, for example. Current research focuses on improving accuracy and efficiency of immersed boundary methods.
- **Multidisciplinary Design Analysis and Optimization (MDAO):** NASA (National Aeronautics and Space Administration) recently conducted a study which outlines the main priorities and concerns for CFD in 2030, entitled *CFD Vision Study 2030* [87]. One of the key goals includes the coupling of various numerical tools. The coupling of, for example, fluid flow solutions and structural deformation simulations, is now possible and desirable. Fluid-Structure Interactions (FSI) are particularly important in maritime CFD for the design of offshore structures and vessels. It is not clear what effects this may have on scalability and performance.

3.4 Summary

The aforementioned NASA study describes multiple expectations of CFD in 2030 – including improvements to physical modelling, MDAO, grid generation, post-processing and knowledge extraction [87]. NASA also calls for dramatic improvements to handle the paradigm-shift in HPC technology; conjointly, they also call for revolutionary algorithmic improvements to the core CFD algorithms in order to efficiently handle larger simulations and more complex hardware. They state that developments in this area have been stagnant for too long. This thesis aims to tackle the cross-over between these last two points, by improving the scalability of the core SIMPLE algorithms (assembly, solve or gradients, for example).

In order to do this effectively, it is important to understand what the future of both HPC and CFD will look like – and this chapter has provided this understanding. The shift towards 'many-core' machines with limited memory bandwidth, heterogeneous architectures, and higher levels of vectorization, will influence design decisions throughout this PhD and may encourage, for example, the use of OpenMP. Furthermore, a shift towards higher-order discretization and linearization techniques is likely to put more pressure on the solve routines – and this should be considered when investigating scalability.

This literature review has been deliberately broad, in order to capture a wide range of future HPC technologies and CFD advancements – some of which reinforce the overall aim of the thesis, and some of which provide direction or understanding for the research that will follow. More focused literature will be discussed in relevant sections throughout this thesis, such as in the following chapter, where a detailed scalability study of ReFRESCO is performed.

Chapter 4

Investigating the Strong Scalability of CFD

This chapter follows the second objective of the thesis, which is to examine the scalability characteristics of ReFRESCO, identifying important bottlenecks and building on insufficient results from the literature. In order to do this, the CFD code is instrumented using profiling tools and hardware counters, and the time spent in various parts of the SIMPLE algorithm is measured. In an attempt to generalize the results and gain a wider picture of the scalability issues facing CFD, permutations of common simulation settings are investigated – including inner-loop convergence tolerances and discretization schemes. This chapter closely resembles and references the work presented in Hawkes et al. [36], available as appendix A, but has been updated with scalability results up to 2048 cores, as in Hawkes et al. [39] – appendix D.

4.1 Introduction

In the literature review, a variety of scalability studies were identified [11, 13, 15, 18, 19, 34, 77]. However, there are a number of deficiencies and disagreements between the literature, and the results are often subjective to the particular code and hardware, which makes it difficult to generalize. This prompts a detailed study of the scalability of ReFRESCO.

In Bhushan et al. [11] the linear-equation system solver, particularly for the Poisson pressure equation, is the main bottleneck to scalability. Culpo [18] reinforces this by considering the main elements of the linear-equation system solver and how they could be improved. However, neither paper investigates the reason for their poor scalability in detail, and neither considers alternative solver algorithms. Gropp et al. [34] provides a good grounding in both these areas, but is now 15 years old – and the conclusions drawn could be somewhat outdated due to changes in hardware.

For commercial CFD codes, such as *Ansys CFX* [19, 42] and *Star-CCM+* [13, 15], the results of scalability tests are not thorough enough to draw conclusions. The tests either show scalability up to a limited number of cores, or hide intra-nodal efficiency losses by normalizing scalability to single-node performance. The details of the test case and mesh size are also not clear. These reports double as marketing material, so it is not in their interest to clearly present scalability issues.

In this chapter, an in-depth review of the scalability of ReFRESCO is conducted on up to 2048 cores. In order to glean information from these tests, profiling tools are used to measure the time spent in key portions of the code. In section 2.1.2, the four most expensive portions of the SIMPLE algorithm were introduced: {assembly, solve, gradients, data exchange}. This logical subdivision was created by analyzing the results of the profiling tools discussed below, and will be used to break down the scalability in a meaningful way.

The results of this scalability study encapsulate years of development of the CFD algorithm, the underlying hardware, messaging libraries (such as MPI [66]), partitioning methodologies [27] and other external tools, such as PETSc [6]; thus should be more relevant than Gropp et al. [34]. As has become apparent during this PhD, it is more-or-less impossible to generalize performance results between different codes and hardware – even when they are fundamentally based on the same algorithm, such as SIMPLE. For example, in EXN/Aero [24], the pressure equation undergoes implicit relaxation, increasing the diagonal of the linear equation-system matrix, thereby making the pressure equation much easier to solve. This removes work from the solve routines, but requires that more outer loops are performed to cancel out the relaxation (thus more time spent in assembly, gradients, data exchange). Differences in linearization and discretization strategies can also have a similar effect.

In an attempt to provide some generalization, various common user settings have been tested. This includes different discretization strategies for convective flux terms, different meshing methods and various turbulence models. For the most part, this had no effect on scalability. Overall, the results appear to correlate well with Bhushan et al. [11] and Culpo [18], showing that the solve routines are the bottleneck to strong scalability. In later chapters, the scalability of the solve routines is investigated in more detail, and solutions are presented.

These results were initially presented in Hawkes et al. [36] (appendix A), but have been updated multiple times throughout this PhD. Shortly after this publication, fundamental changes were made to the data exchange routines in ReFRESCO which dramatically improved their scalability – updated results were presented in Hawkes et al. [39] (appendix D). In both the aforementioned studies, the scalability results were performed up to 512 cores. For the majority of this PhD, all experiments were limited to 512 cores due to the default permissions on Iridis4. At the end of this PhD, a special opportunity was granted to test on 1024/2048 cores – some additional results are presented here to reflect this, but it was not possible to re-run all previous experiments.

4.2 Methodology

The scalability of ReFRESCO is measured on Iridis4, from 1 through to 512 cores (and up to 2048 in section 4.3.5). Two test cases, both of approximately 2.7-million cells, are used to evaluate the scalability of the code and algorithms. Various tools have been used to *instrument* the code – measuring the performance of the code in a non-intrusive way. These are introduced below.

4.2.1 Test Cases

Two test cases are used in these experiments, and will be used throughout this thesis. The first is a laminar-flow, canonical, unit-length, cubical, three-dimensional lid-driven cavity flow (LDCF). A uniform structured mesh of 2.69-million cells (139^3) is used, and only momentum and pressure equations are solved. The simulation mimics an infinite domain, with two cyclic boundary conditions. The remaining four boundaries are constrained with Dirichlet boundary conditions, one of which specifies a tangential, non-dimensional velocity of 1. The domain is shown in figure 4.1. The Reynolds number is 1000.

The second test case is the Kriso Very Large Crude Carrier (KVLCC2) double-body wind-tunnel model. This test case is chosen as it is a well-known maritime CFD test case, and is used for a variety of CFD workshops – and is backed up by experimental model-testing [57]. The mesh is a three-dimensional, multi-block structured mesh consisting of $2.67m$ cells. A k-ω, two-equation shear stress transport turbulence model is used [63]. The domain and mesh are shown in figures 4.2 and 4.3 respectively. The Reynolds number is 4.6×10^6.

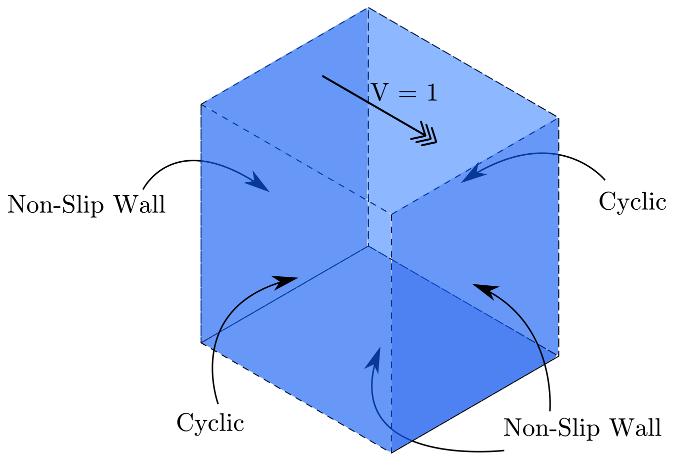


Figure 4.1: The 3D lid-driven cavity flow (LDCF) domain, with three stationary walls, two cyclic boundaries and one velocity-forcing wall. All edges of unit length.

The two test-cases are designed to have a similar number of cells and the size of the mesh has been chosen to show the full range of scalability issues. On 512 cores, the cells-per-core ratio reaches approximately 5200 which helps demonstrate the parallel bottlenecks on a large number of cores; yet the problem is substantial enough to show memory bandwidth issues on a single node. On 2048 cores, the cells-per-core ratio reaches 1300. An initial study was performed on meshes ranging from $317k$ to $23.5m$ cells in order to determine the most useful mesh size.

It is important to note that whilst the two meshes are structured, ReFRESCO treats them as unstructured meshes and does not gain any direct computational benefits of a

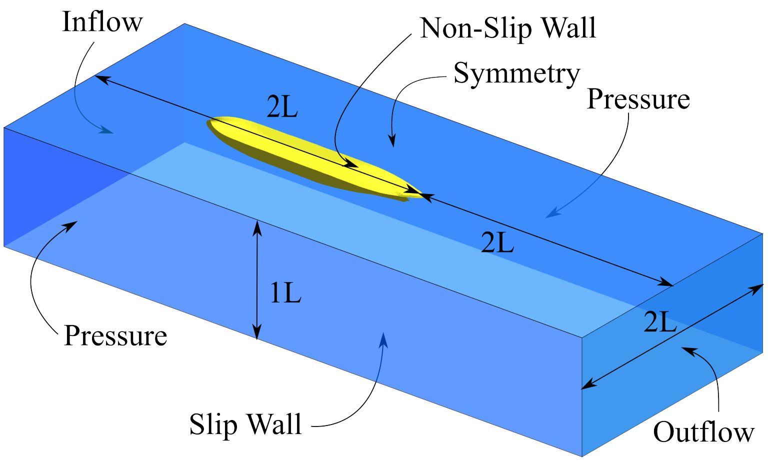


Figure 4.2: The domain used for the KVLCC2 double-body wind-tunnel simulation. Symmetry boundary conditions are applied at the water-plane, but port- and starboard-sides of the centre-line are both simulated.

structured memory layout. However, in order to observe any indirect effects of mesh structure (such as coincidentally improved memory bandwidth or reduced stiffness in the linear equation-systems due to improved mesh geometry), scalability of a truly unstructured mesh has been compared to the structured mesh described. The results are discussed in section 4.3.3.

For scalability tests, the CFD solver is not executed to convergence – it would simply be too expensive. Initially, scalability studies were performed over 400 outer-loops; however, this was reduced to 100 outer loops when experiments began on 1024/2048 cores. On 1024/2048 cores, each case required approximately 6 minutes of wall-time; allowing 28 permutations of the solver to be tested in the 3 hours that were available. It is generally observed that the equation systems are stiffer at the start of a simulation, where the flow is uninitialized, but the effects were found to be negligible in these particular test cases. Thus, unless specified, all of the scalability experiments in this thesis have been (re-)run with 100 outer loops.

An inner-loop relative convergence tolerance (ILCT) of 0.1 is used for momentum, pressure and turbulence equations. ILCT$=0.01$ and 0.001 were also tested in section 4.3.1, in order to cover a common range of ILCT settings used with ReFRESCO.

Relaxation is applied to all outer-loops to stabilize the non-linear iterative process. For momentum and turbulence, $\alpha_{exp}=0.15$ and $\alpha_{imp}=0.8$, ramping up to $\alpha_{imp}=0.85$ linearly over the first 100 iterations. For the pressure equation, $\alpha_{exp}=0.1$ and no implicit relaxation is used.

QUICK with flux limiters [44] is used to discretize the convective terms in the momentum equations. This scheme is second-order, reducing to first-order where flux gradients are high. A first-order upwind scheme is used to discretize the convective terms of the turbulence equations. The effects of changing the discretization scheme will be observed in section 4.3.2. It may be that higher-order terms impact the cost of assembly routines, or impact the linear equation-systems and solve routines. For all equations, the diffusion terms are discretized using a 2nd-order central scheme. The

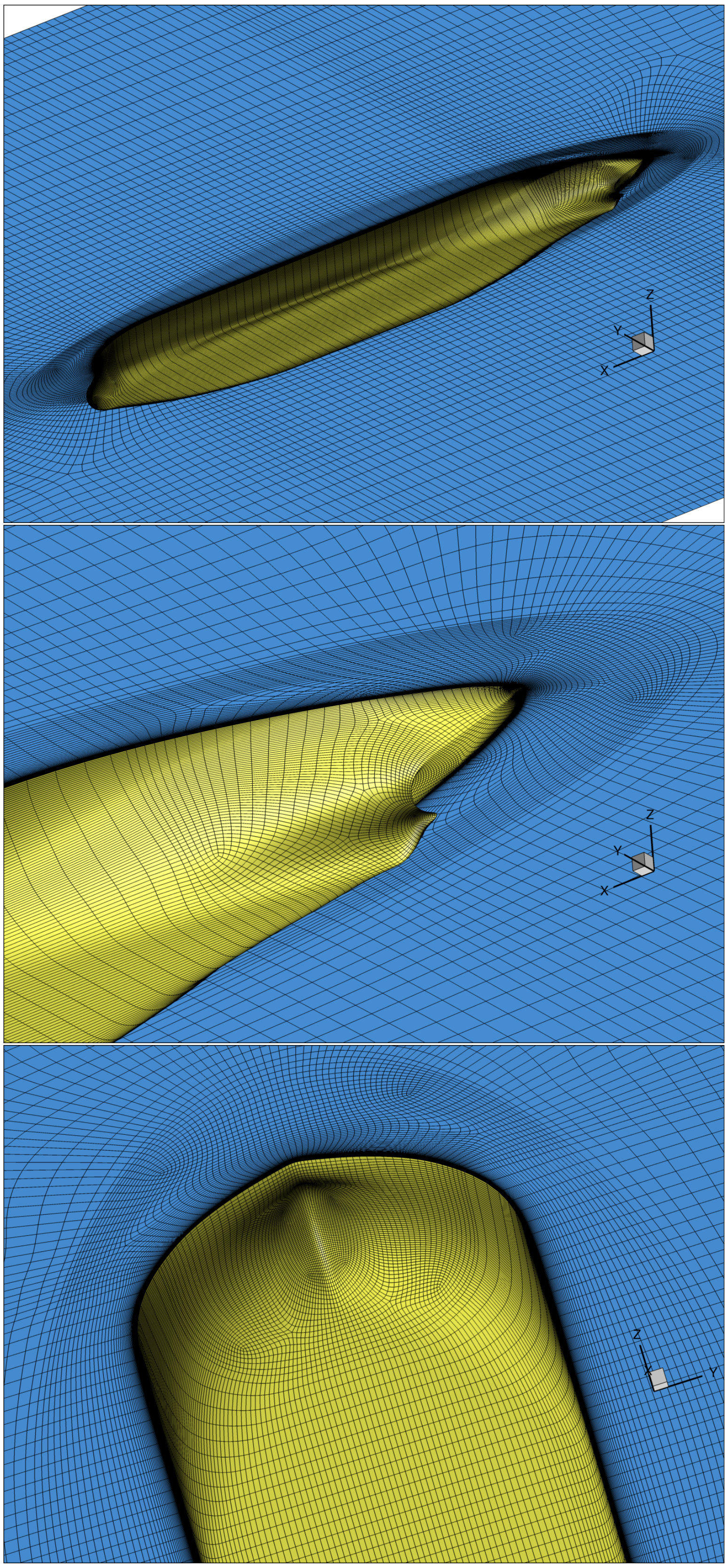


Figure 4.3: The KVLCC2 mesh is a multi-block structured mesh consisting of 2.67-million cells.

gradients of any variable (velocity, pressure, turbulence quantities) are discretized using a 2nd-order Gauss-theorem scheme.

For the solution of the linear equation-systems (for all equations), a GMRES (Generalized Minimal Residual method) solver is used with a right-side Block Jacobi preconditioner, from PETSc [6], as in the scalability study of Gropp et al. [34]. The choice of linear solver has a very large impact on the scalability of the solve routines, and a more thorough investigation is performed in chapter 5.

4.2.2 Instrumentation

In order to observe how the assembly, solve, gradients and data exchange routines scale, ReFRESCO has been profiled using *Score-P* [101]. Score-P is a compile-time wrapper which automatically logs various run-time events, such as function calls and durations. It has been carefully filtered such that only critical functions are measured, and the effect on total run-time is less than 2%. Score-P can be linked to PAPI (Performance Application Programming Interface)[12] which gathers additional information from physical hardware-counters.

Two hardware counters are enabled for these tests. The first measures FLOPS, which helps to determine whether the processing units are saturated. For an unstructured CFD code this is an unlikely situation, as the indirect memory access tends to delay the processor. Structured-mesh codes are more likely to reach these limits due to vectorization of memory operations.

It would be ideal if memory bandwidth could be measured, however, it is not possible to measure this directly. Instead, one can measure *cache misses* – any occurrence of when a CPU instruction is halted and required to wait for a memory transaction. Level one (L1) cache misses give a good indication of memory-fetching issues (since a cache miss must result in a memory or next-level cache transaction). However, there are two issues. Firstly, the compiler will often pre-fetch memory into the cache, resulting in memory bandwidth usage which is not detected by this hardware counter. Secondly, an L1 cache-miss will be registered even when the data resides in L2 or shared cache (which is still very fast compared to off-chip memory).

Due to these imperfections, it is often difficult to use the hardware counters to draw concrete conclusions. However, they can assist in understanding scalability bottlenecks which are revealed by ordinary profiling, particularly at the shared-memory level. Other hardware counters are available (such as L2 cache misses) on Iridis4, but they cannot be enabled concurrently due to limited register sizes and competing circuitry.

4.3 Scalability Tests

The results of the initial scalability tests are shown in figure 4.4. These scalability graphs are used throughout this thesis in various forms, so it is worth explicitly explaining their format. The graphs show the scalability of various routines from 1 through to 512 cores. Let T_C be the time required for any given routine on C cores, thus T_1 is the serial run-time. A scalability factor (S) can be determined for any C, where:

$$S = \frac{T_1}{T_C}$$

thus the scalability at $C = 1$ is one; and ideal scalability is a linear relationship where $S = C$. When normalized to serial run-time (T_1) in this way, this factor is often called the parallel speed-up factor, but the more generic term is preferred here. Parallel efficiency, often given as a percentage, is defined as $\frac{S}{C}$ where 100% parallel efficiency represents ideal scalability.

When comparing the scalability of two fictional routines, $R1$ & $R2$, the normalization of each routine to T_1 hides the absolute time spent in each routine, thus hiding an important part of the result. Regardless of scalability, if $T_{1,R1} >> T_{1,R2}$ then this may be an overriding factor in determining which routines are the greatest bottleneck to exascale computing, since any scalability losses in $R1$ will be amplified by their relative cost.

In the scalability plots, an embedded bar-chart is provided which shows the absolute (not normalized) time spent in each routine. This is plotted as $C \times T_C$ (core-hours), and is shown for various C. With ideal scalability ($S = C$), $C \times T_C$ should remain constant. It is often helpful to see this scalability information duplicated in this view, and the relative differences between routines can be quantified.

Figure 4.4.a and 4.4.b show the scalability of the code as the number of cores increases. The results from the hardware counters are shown in figure 4.5, for the LDCF test case.

Total scalability is poor overall, suffering significantly on just 16 cores due to intra-nodal bottlenecks and worsening to the point at which almost no speed-up is gained from adding additional nodes. On 512 cores the parallel speed-up is just 128 (KVLCC2) and 100 (LDCF). This scalability is similar to other codes, such as OpenFOAM [18, 77], STAR-CCM+ [13] or Ansys Fluent [HP 2014]. Exact comparisons between various codes on identical hardware were not feasible.

The routines outlined earlier {assembly, solve, data exchange and gradients}, for each equation in each outer loop, account for the large majority of overall run-time. The remaining time (other) is spent (mostly) in one-off functions such as file IO or MPI initialization, thus is subjective to the length of the simulation and amount of IO required. These other routines may also increase significantly if additional features such as moving, deforming and adaptive grids are used – again, this is highly subjective.

The assembly and gradients routines scale favourably, reaching almost 90% parallel efficiency. The high, and increasing, cache-miss rate of the gradients routines is curious, as it does not seem to affect performance (FLOP rate is maintained). It is likely that the data required resides in L2 or shared-cache, rather than off-chip memory, so the impact of these L1 cache misses is much lower.

The data exchange routines are not visible in the scalability plots, because normalizing against ($T_1 \approx 0$) gives inverse scalability (no communications are required in serial operation). In reality, these routines scale reasonably – as the number of cores

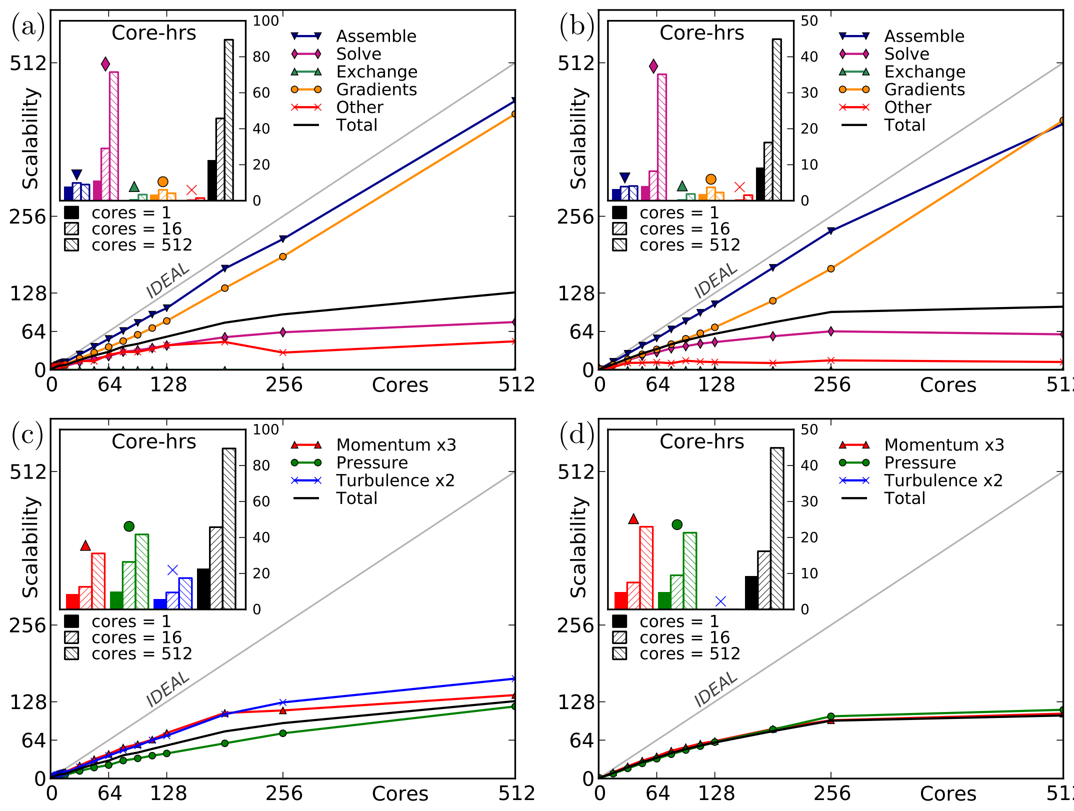


Figure 4.4: Scalability of the code, and the profiled routines within, as the number of cores increases. These results used GMRES as the linear solver with a Block Jacobi preconditioner, and an inner-loop relative convergence tolerance of 0.1. (a) KVLCC2 breakdown by routine, (b) LDCF breakdown by routine, (c) KVLCC2 breakdown by equation, (d) LDCF breakdown by equation.

increases the size of the messages become smaller, and these messages can be sent concurrently. With inadequate load-balancing these data exchanges could become costly, due to the implicit synchronization of MPI processes. However, the results show that these communications account for a small proportion of overall run-time.

As consistent with literature, the solve routines have poor scaling and are a major contributor to total run-time, thus are the main concern for scalability.

For the solve routines, the hardware performance counters show a high cache-miss rate between 16 and 128 cores, corresponding to saturated memory bandwidth at the intra-nodal level. Memory-bandwidth-per-node has been growing at approximately half the rate of processing power leading to problems with memory bandwidth [85]. In Gropp et al. [34], conducted in 2000 on single-core processors, memory-bandwidth problems were apparent but not as concerning – changes in architecture over the last decade have made memory bandwidth issues more critical.

Beyond 128 cores, it becomes difficult to utilize the hardware-counter results for more meaningful analysis. Scalability worsens at this point, and cache misses in the solve routines become less frequent as FLOP rate continues to decrease. This is due to the oft-observed global communication bottleneck, which overtakes memory bandwidth as the main problem with the solve routines. An illustration of a single GMRES iteration is

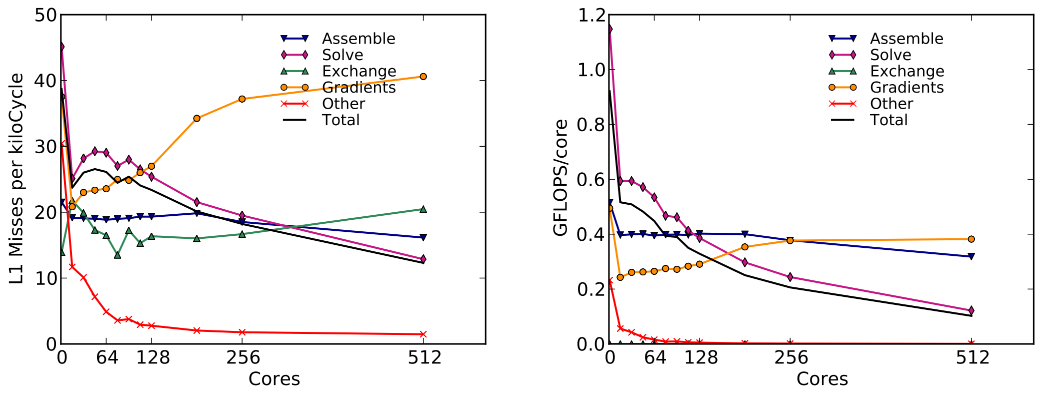


Figure 4.5: An example of the information gleaned from PAPI hardware counters. For all routines in the LDCF test case, the number of first-level (L1) cache-misses per thousand clock cycles [left] and the total floating-point operation rate (FLOPs) [right] are shown.

shown in figure 4.6. This pattern includes sparse-matrix-vector-multiplication (SpMV), involving scalable neighbour-to-neighbour communications; and two global reduction-broadcast routines which require a hierarchical global communication (scaling poorly with $T_C \propto \log_2(C)$). These global communications create a scalability bottleneck when a high number of cores are used, and has been well-documented in the literature [18].

To demonstrate this further, results are shown in figure 4.7, showing the scalability of a single GMRES iteration. The SpMV routines are shown to scale well, whilst the remainder (including global communications) scale poorly. This kind of detailed analysis can be performed with Score-P or the built-in profiling features of PETSc, and is used throughout this PhD to provide more qualitative insight into the scalability of linear solvers.

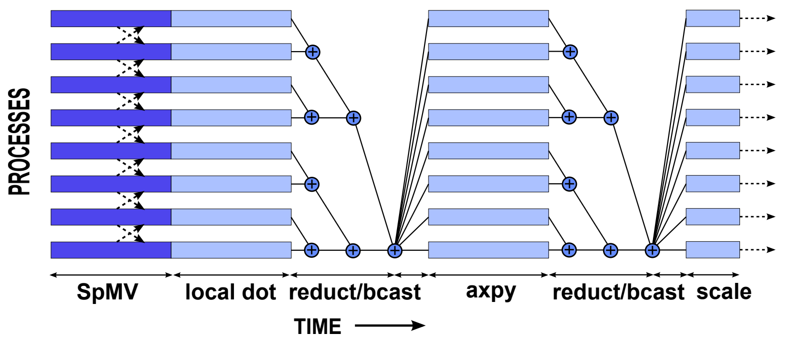


Figure 4.6: An illustrative trace of a GMRES iteration on 8 cores (not to scale). Note the local neighbour-to-neighbour communications performed asynchronously in the SpMV routine, and the two reduction-broadcast patterns. There are many variations of the reduction-broadcast algorithm which cannot be illustrated clearly. The most common is the 'butterfly' algorithm which completes in $\log_2(C)$-time, combining the reduction and broadcast into a single hierarchy of latency-bound messages. As the number of cores increases, this reduction-broadcast takes longer, whilst other routines take less time.

Figure 4.4.c and 4.4.d shows the breakdown of time spent in the various equations (pressure, momentum, turbulence). In both cases, the single pressure equation took

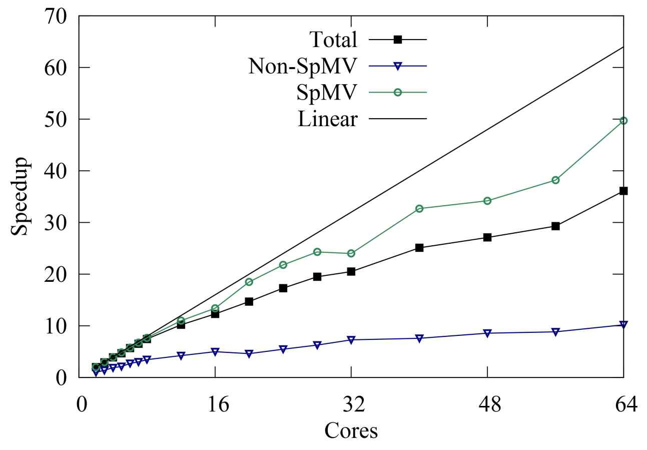


Figure 4.7: Profiling of SpMV and Reduction routines within a benchmark test of GMRES.

considerable time to compute – similar to all three momentum equations combined. In incompressible-flow simulations the pressure equation is much harder to solve than other transport equations, due to its elliptic Poisson form. Overall, the solve routines appear to be the biggest concern for scalability; but in order to help generalize these results, a variety of control parameters are tested in the following sections.

4.3.1 Inner-Loop Convergence Tolerance

It is often desired to run the CFD solver with a stricter inner loop convergence tolerance (ILCT), as this allows less outer-loop relaxation to be used and may also be required in order to converge particularly challenging simulations. A brief investigation was performed using the KVLCC2 test case to determine the effects of changing ILCT in a practical simulation, before considering the effect on scalability.

In these tests, ILCT is varied between 0.5 and 10^{-6}, starting from a restarted solution (of 100 outer loops) with the default settings (described above). Two outer loop relaxation schemes are tested: $\alpha_{imp} = 0.9$ & $\alpha_{exp} = 0.1$ (tight), and $\alpha_{imp} = 0.8$ & $\alpha_{exp} = 0.2$ (relaxed). In this way, the computational effort associated with linearity and non-linearity of the transport equations can be changed by shifting the focus to the inner loop or outer loop respectively. The total time to reach 10^{-5} L_∞-norm outer loop convergence was observed. The L_∞-norm was used (absolute maximum residual) instead of the L2 norm (least-squares residual), as it is stricter and ensures no specific parts of the flow are under-converged when the ILCT is increased. The study was performed using $C = 16$ in order to minimize scalability problems.

Figure 4.8 shows the results of this study. In general, a stricter inner loop convergence tolerance provides no benefit to overall convergence and serves only to increase the time-per-iteration except in the case where the linear set of equations is not solved well enough to allow the non-linear iterations to converge. The best-case scenario is the relaxed scheme with an ILCT of 0.1, which is approximately 50% faster than the tighter scheme at its optimal ILCT of 0.01. Nonetheless, it is often necessary to use a stricter ILCT, particularly for the pressure equation, and the effects on scalability should be documented.

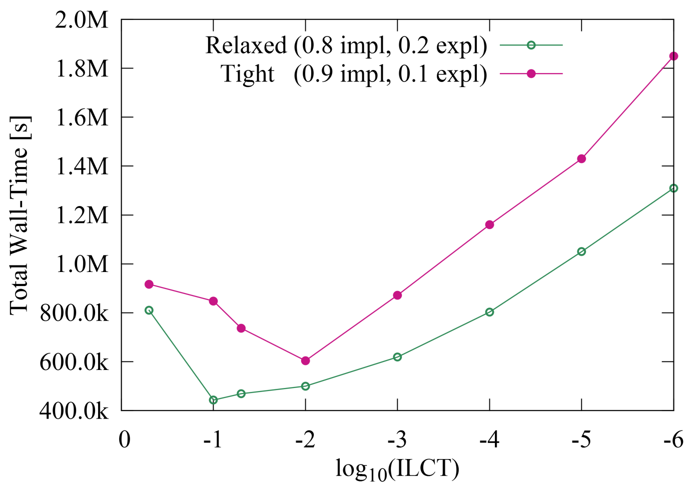


Figure 4.8: Effects of ILCT on total wall-time using two different outer loop relaxation factors.

The scalability tests from the previous section were repeated using a tolerance of 0.01 and 0.001 (see figure 4.9). As one would expected, this results in a far greater time spent in the solve routines, particularly for the pressure equation. This provides further motivation for improving the strong scalability of the solve routines. LDCF is not shown here, but the results were similar.

4.3.2 Discretization Schemes

The previous scalability studies have been performed with a QUICK momentum convective discretization scheme with a flux limiter. Here, this discretization scheme is varied to ensure that the scalability was independent of the chosen scheme. A first order upwind (UD1), a 50%-central-difference-50%-upwind blend (CD5-UD5), a 90%-central-difference-10%-upwind blend (CD9-UD1), QUICK with limiter and QUICK without limiter (QUICK-NL) were tested. This study was performed early in the PhD, and does not encapsulate recent performance improvements to ReFRESCO. This is unlikely to affect these results in a significant manner. This study was performed only for the KVLCC2 test case, and uses 400 outer loops.

Firstly, the difference in scalability is demonstrated in figure 4.10.a for a subset of these discretization schemes. There is no discernible difference between the different schemes in terms of performance (not shown) and scalability. A full scalability assessment was not performed with QUICK-NL and CD5-UD5, it is assumed that scalability was similar. Due to the methods used by ReFRESCO (particularly *deferred correction*, see section 2.1.2), the stencil of the linear equation system does not change regardless of the order of the discretization. This means that it is unlikely to have any effect on the already-poor solve routines. This is also an encouraging result for higher-order Discontinuous Galerkin methods, as discussed in chapter 3. These methods also do not increase the stencil size of the linear equation systems, although they do create other

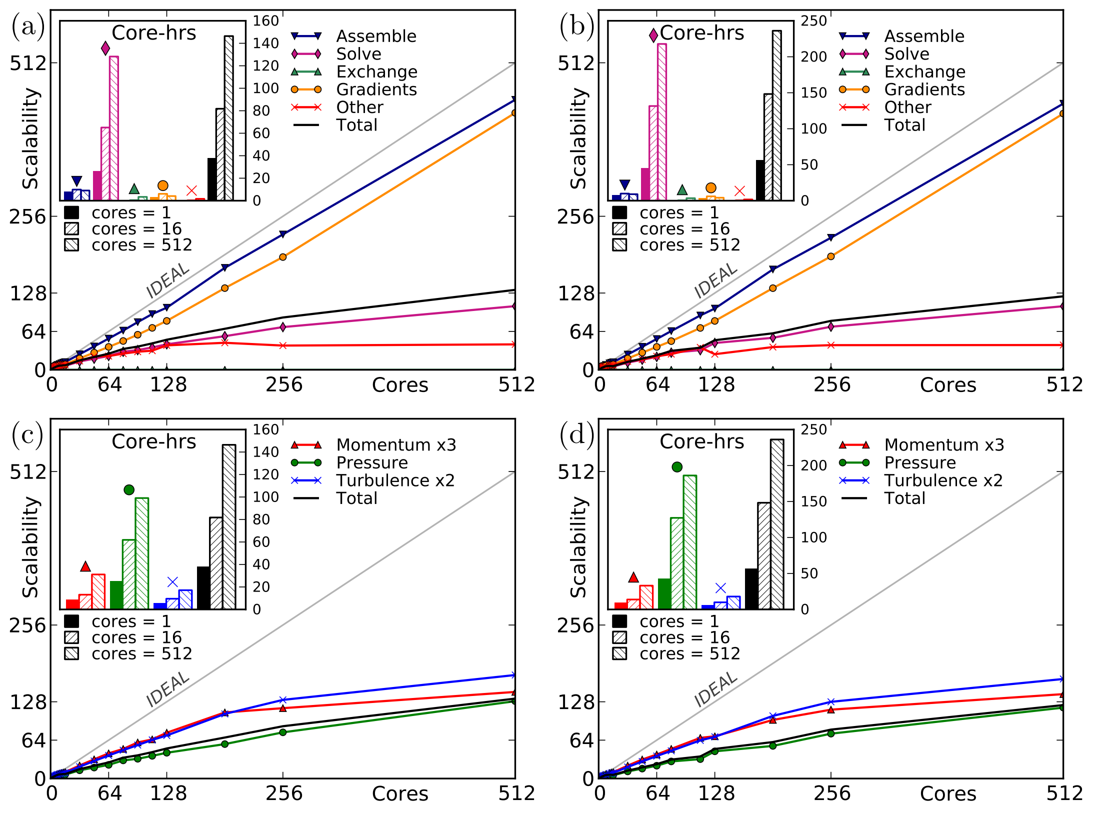


Figure 4.9: Scalability of the code with an inner-loop convergence tolerance of (a, c) 0.01 and (b, d) 0.001 for the KVLCC2 test case as in figure 4.4, showing scalability of (a, b) the various routines and (c, d) the various equations, using GMRES with a Block Jacobi preconditioner. The results show poor scaling of the solve routines, which particularly influences the cost of the pressure equation. Similar results were obtained for the LDCF test case.

issues for the linear solver [69, 72].

Although somewhat unrelated, a grid convergence study has been conducted using a range of grids from 317k to 23.4m cells. The solution was allowed to reach an outer loop L2-norm convergence of 10^{-5}. The computed form factor $(1 + k)$[65] is compared to wind-tunnel experimental results [57] in figure 4.10.b. The results are presented against the grid coarsening factor, which is the ratio of cells compared to the finest grid.

As expected, the higher-order schemes provide much higher levels of accuracy, with QUICK matching CD9-UD1 using far fewer cells (1.11m vs. 10.0m). The importance of higher-order methods such as Discontinuous Galerkin methods is clearly not to be underestimated.

4.3.3 Unstructured vs. Structured Meshes

Structured grids are often not feasible for practical applications, where the labour cost of creating the grid is too high. Thus it is important to measure the scalability differences between structured- and unstructured-grid simulations. Internally, ReFRESCO treats the two identically, and does not have the advantages of a solver designed solely for structured meshes. However, grid quality may affect the smoothness of the linear

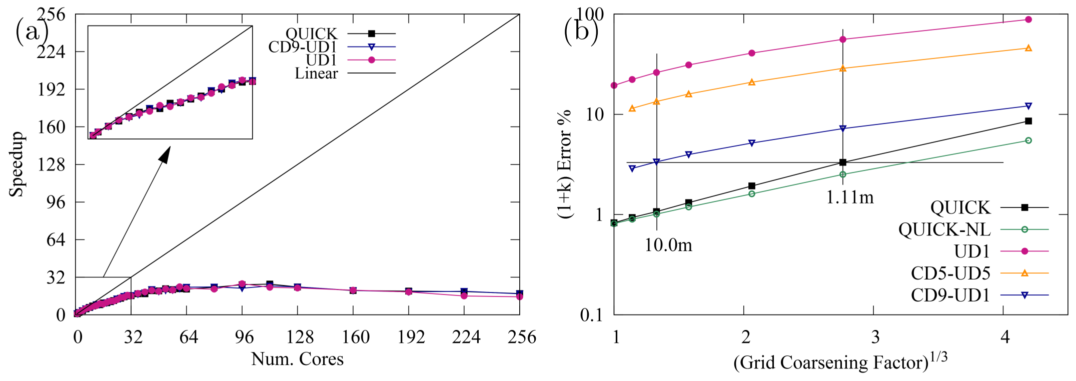


Figure 4.10: (a) Total scalability of ReFRESCO using various convective schemes over 400 outer loops. (b) Grid convergence study and verification for the KVLCC2 test case using various convective discretization schemes; showing form factor error as a percentage, compared to experimental results.

equation-systems, and some pseudo-structure may have advantages for memory access. Unfortunately, it was not possible to create two grids of exactly the same number of cells. An unstructured KVLCC2 grid of 12.5m cells was used, and the scalability of two structured grids (10.0m and 15.8m) was linearly interpolated to obtain comparisons. This method is not the most accurate, but the results showed *no noticeable difference in scalability or absolute performance*, suggesting that mesh structure had negligible impact. The reader is referred to appendix A for further details and results.

4.3.4 Turbulence Models

Similarly, the effect of choosing different turbulence models was investigated. The default two-equation $k - \omega$ shear-stress transport (SST 2003) model [63] was tested against a two-equation KSKL model [62] and a one-equation Spalart-Allmaras (SA) model. The reader is referred to appendix A again, for further details and results, however, these results are somewhat out-dated. The results favoured the one-equation SA model due to lower absolute wall-time and better scalability; but these differences were amplified by inefficiencies in the data exchange routines, which have since been corrected. It is not expected that scalability concerns will determine which turbulence model should be used – the physical properties of these models are more important – so these experiments have not been re-run. It is, however, important to note that the inclusion of other transport equations – such as those for cavitation, may have a larger effect on total scalability. These models typically add additional terms to the pressure equation, affecting the solve routines. Additionally, some models, such as two-equation turbulence models, require additional parallel communications – for example, to compute eddy viscosity.

4.3.5 1024/2048-Core Scalability Experiments

The scalability studies from section 4.3, earlier in this chapter, were repeated on 1024 and 2048 cores. The results are shown in figure 4.11. These scalability studies clearly validate the conclusions drawn in section 4.3; showing a further degradation of the solve routines, especially in the pressure equation. On 2048 cores, the parallel efficiency of the solve routines drops to just 2.1%, and occupies 95% of the total wall-time. Some problems are also noted in the assembly routines – due to oft-overlooked communications required to update boundary conditions – but there was no opportunity to investigate this further on this number of cores, due to the complexities of scheduling these large experiments.

Scaling to this level is likely to require careful optimization of parallel communications (for example, the assembly communications could easily be overlapped and hidden), as well as improvements at the algorithmic level. ReFRESCO has never been tested on this many cores before, and there are many minor improvements and optimizations that should be made before this becomes a practicality. Usually, these optimizations are of little academic interest and are therefore outside the scope of this PhD.

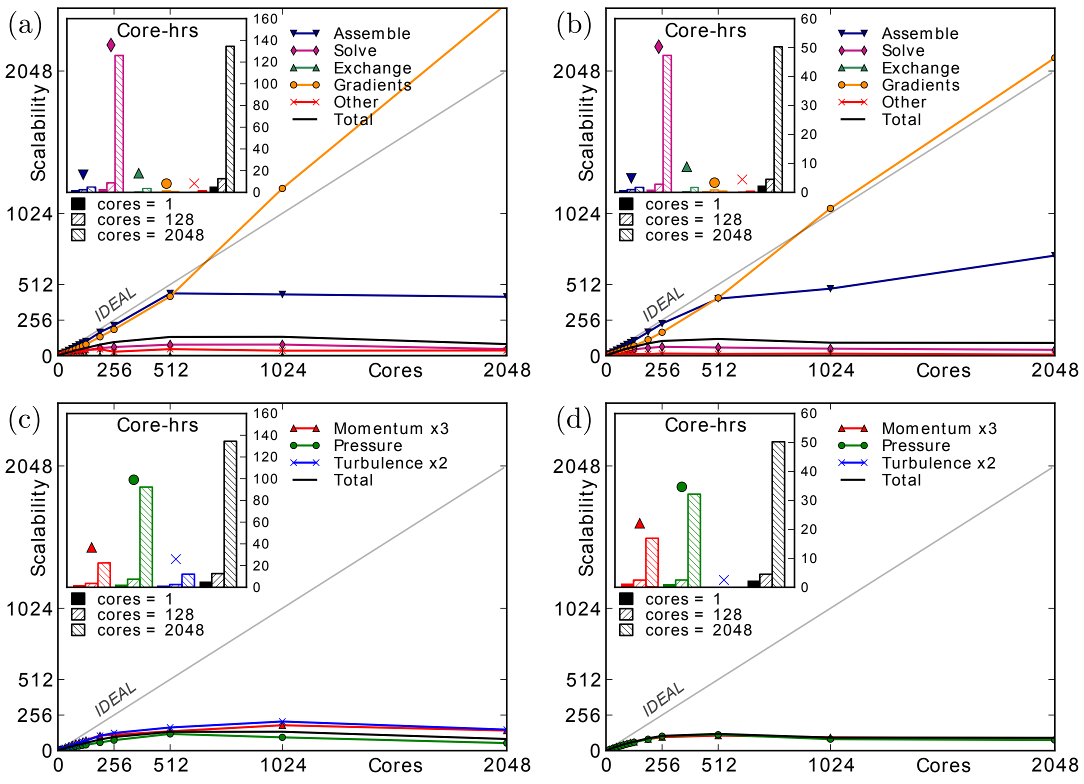


Figure 4.11: Scalability of the code, and the profiled routines within, as the number of cores increases up to $C = 2048$. These results used GMRES as the linear solver with a Block Jacobi preconditioner, and an inner-loop relative convergence tolerance of 0.1. (a) KVLCC2 breakdown by routine, (b) LDCF breakdown by routine, (c) KVLCC2 breakdown by equation, (d) LDCF breakdown by equation.

4.4 Summary & Conclusions

Throughout this chapter, the solve routines, particularly for the pressure equation, have revealed themselves as the least scalable and most expensive part of the CFD algorithm, achieving a parallel efficiency of just 2.1% on 2048 cores, and occupying 95% of total walltime. There are likely to be other scalability bottlenecks, especially when considering more advanced features such as adaptive meshing. However, these concerns are subjective to the particular CFD code or test case, whereas the unscalable pressure-equation solvers will be an overarching, core problem. Furthermore, the scalability of the solve routines has been shown to be mostly independent of control parameters such as discretization scheme and mesh structure, and are likely to remain a concern in the future.

In the simple test cases used so far, performance is optimum with an inner-loop convergence tolerance of just 0.1, but more complex test cases require much better convergence (as will be shown in chapter 8). This is particularly true when considering some of the new CFD methodologies that are likely to become prevalent, such as additional transport equations, higher-order linearization and discretization, and much higher fidelity simulations (enabling LES, for example). As was shown in this chapter, the scalability problems of the solve routines are amplified even further as the ILCT is reduced to 0.01 or 0.001. Indeed, on just 512 cores the solve routines occupy 92% of total wall-time with an ILCT of 0.01.

Although this scalability bottleneck is severe, it does at least provide a clear direction for researchers aiming to improve the scalability of CFD. Asides from relatively uninteresting optimizations required to scale to 2048-cores, the assembly, gradients and data exchange routines all scale well.

Thus far, only one linear solver has been tested – a Krylov Subspace solver (GMRES) with Block Jacobi preconditioning. Investigation into scalable Krylov Subspace methods is a very active area of research [31, 61, 108]; as is research into other types of solver, such as multigrid methods [3, 29] or asynchronous methods [2, 5, 10]. This research encompasses a wide range of numerical sciences (not just CFD), and there are often complexities in applying these solvers specifically to CFD. However, it may be that the scalability concerns investigated in this chapter can be circumvented or diminished by using alternative linear solvers.

Investigation into alternative solvers is the subject of the following chapter. Indeed, the remainder of this thesis is focused on discovering new, scalable methods, to solve the pressure equation efficiently.

Chapter 5

Investigating Alternative Linear Solvers and Preconditioners

In this chapter, the theory behind linear equation-system solvers is explored – covering simple *stationary* solvers such as Jacobi and Gauss-Seidel; and moving on to *non-stationary* methods such as Krylov Subspace methods and multigrid methods. *Preconditioning* of the equation-systems is also discussed. Various solvers and preconditioners are then tested, in an attempt to circumvent the scalability problems identified in the previous chapter. Furthermore, this chapter will investigate the reasons for the poor scalability of the various linear solvers and preconditioners, by using the instrumentation tools (Score-P and PAPI) in more detail. This will provide insight into the scalability problems and guide further research. This work was first presented in Hawkes et al. [36], and later updated in Hawkes et al. [39], available as appendices.

5.1 Introduction

The task of the linear solver, for each transport equation in each outer loop, is to solve:

$$\mathbf{Ax} = \mathbf{b}, \tag{5.1}$$

where $\mathbf{x}$ is the unknown solution vector, $\mathbf{A}$ is an N-by-N sparse coefficient matrix, $\mathbf{b}$ is the constant right-hand-side (RHS) vector, and N is the number of elements.

There are a number of methods that can be used to solve this system. The most basic, so-called *stationary* methods will be considered first.

Henceforth, the term *linear solver* or *solver* is used to refer to linear equation-system solvers – not to be confused with the overarching CFD solver.

5.1.1 Stationary Solvers

Beginning with an initial guess for $\mathbf{x}$, the system can be solved iteratively:

$$\mathbf{x}_k = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}_{k-1} + \mathbf{D}^{-1}\mathbf{b} \tag{5.2}$$

where $\mathbf{D}$ is the diagonal of $\mathbf{A}$, and $\mathbf{L}/\mathbf{U}$ are the lower- and upper-triangles respectively. The notation k represents the iteration number. This is the Jacobi method, and the matrix $\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})$ is the iteration matrix, $\mathbf{M}$. Each equation (from 1 to N) can be solved (*a.k.a.* relaxed) independently as follows:

$$x_{i,k} = \left(-\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_{j,k-1} + b_i \right) / a_{ii}, \quad i = 1, \ldots, N. \tag{5.3}$$

where a, x and b are the individual components of $\mathbf{A}$, $\mathbf{x}$ and $\mathbf{b}$ respectively. This algorithm is trivial to parallelize, since each individual relaxation in the system can be performed independently in a process or thread. However, at the end of each iteration the new values of $\mathbf{x}$ must be communicated before the next iteration can begin, and this introduces an implicit global synchronization – no process or thread can become more than one iteration out of sync with its neighbours. Each Jacobi iteration can be considered as a sparse-matrix-vector-multiplication, which has been shown to scale well (figure 4.7).

Numerically, improvements to this scheme can be made by using values from the current iteration as they are available. This is the Gauss-Seidel method, and is rarely used due its strictly sequential operation (since the equations must be performed in order).

$$\mathbf{x}_k = -(\mathbf{D}-\mathbf{L})^{-1}\mathbf{U}\mathbf{x}_{k-1} + (\mathbf{D}-\mathbf{L})^{-1}\mathbf{b} \tag{5.4}$$

A variation on lecixographical Gauss-Seidel is a parallel Block Gauss-Seidel, where each process owns a block of the matrix. Each block undergoes Gauss-Seidel internally, with Jacobi-style updates at block boundaries [98, chap. 5]. 'Gauss Seidel' is often used to refer to parallel Block Gauss-Seidel whenever a parallel implementation is implied .

An extension of the Gauss-Seidel method, called Successive Over-Relaxation (SOR), adds a scalar magnification to each relaxation for faster convergence. When $\omega = 1$ SOR is equivalent to Gauss-Seidel, and it is common to use the term SOR to refer to Gauss-Seidel or Block Gauss-Seidel solvers.

$$\mathbf{x}^k = -(\mathbf{D}-\omega\mathbf{L})^{-1}[\omega\mathbf{U} + (1-\omega)\mathbf{D}]\mathbf{x}^{k-1} + (\mathbf{D}-\omega\mathbf{L})^{-1}\omega\mathbf{b}, \quad 0 < \omega < 2 \tag{5.5}$$

These methods are known as *stationary methods*, because their formulation does not change between iterations. For all stationary methods, to determine whether the solution is converged, the ℓ^2-norm of the residual vector ($\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$) must be computed. This requires an explicit global communication. It is common to compute a residual periodically (for example, every 100 iterations) in order to avoid unnecessary communications. A more thorough description of stationary methods can be found in Barrett et al. [8].

Numerically, the Jacobi method, or its derivatives such as Gauss-Seidel or Successive Over-Relaxation methods, are weak. Their convergence rate is limited by the *spectral radius* of the iteration matrix, $\rho(\mathbf{M})$, which is dependent on the matrix size and stiffness.

5.1.2 On The Spectral Radius

The linear-equation systems arising from CFD contain various frequencies of error, which must be removed. High-frequency errors, corresponding to low wave-numbers and low eigenvalues, can be quickly *smoothed* using simple solvers. Low-frequency errors, corresponding to high eigenvalues, are much more difficult to remove. In ReFRESCO, the momentum and turbulence equations do not contain high-frequency errors. This is usually the case in SIMPLE-based CFD solvers, depending upon the linearization strategy. However, the pressure equation is an elliptic *Poisson* equation; meaning that the lowest frequency errors traverse the entire domain. High-frequency errors can be considered local errors, reduced quickly by simple relaxations. Low-frequency errors can be considered global errors, requiring a more global solution – and, often, global communications.

The maximum eigenvalue of the iteration matrix $\mathbf{M}$ is known as the spectral radius, $\rho(\mathbf{M})$, and, for Poisson equations, it approaches 1 as N increases [103]. For stationary methods, the error in the lowest frequency cannot be reduced faster than $1 - \rho(\mathbf{M})$ per iteration, and $\rho(\mathbf{M}) < 1$ is a necessary condition for convergence. Higher-frequency errors can be reduced more quickly, allowing faster convergence, but the convergence properties of stationary methods eventually deteriorate with $O(N^2)$. Since the size of the equation-system also increases with $O(N)$, the total work required by stationary methods scales with $O(N^3)$. Over-relaxation with SOR does not help, because $\omega\rho(\mathbf{M}) < 1$ must be satisfied for convergence, thus $\omega > 1$ is infeasible for the solution of the pressure equation.

5.1.3 Non-Stationary Solvers

It is possible to add a form of 'memory' to a linear equation-system solver, where some information about the solution path is stored in a vector subspace, known as the Krylov Subspace (KSP). For example, in the popular (Bi-)Conjugate Gradient Squared (BCGS) algorithm, the solver remembers the previous gradients that it followed. By searching orthogonally to previous gradients, the solver becomes smarter and more effective. In GMRES (Generalized Minimal Residual Method) the subspace is based on the Arnoldi iteration, and is used to find an approximate solution based on the minimal residual of these vectors [8].

These non-stationary KSP methods are not limited by spectral radius, thus continue to reduce low-frequency errors when N increases. Figure 5.1 shows a comparison between a KSP (GMRES) and a non-KSP solver (Block Gauss-Seidel), illustrating the effect this has on the number of iterations to convergence. The equation-systems being solved are pressure equations extracted from a two-dimensional lid-driven cavity flow simulation. Note the initial grace period, where high-frequency errors are rapidly smoothed.

Though numerically powerful, the KSP methods are computationally inefficient on a larger number of cores due to global communication patterns, as discussed in chapter 4 and figure 4.6. Improvements have been made to GMRES, in the form of 'pipelined' GMRES (P-GMRES) which attempt to overlap the first reduction with serial

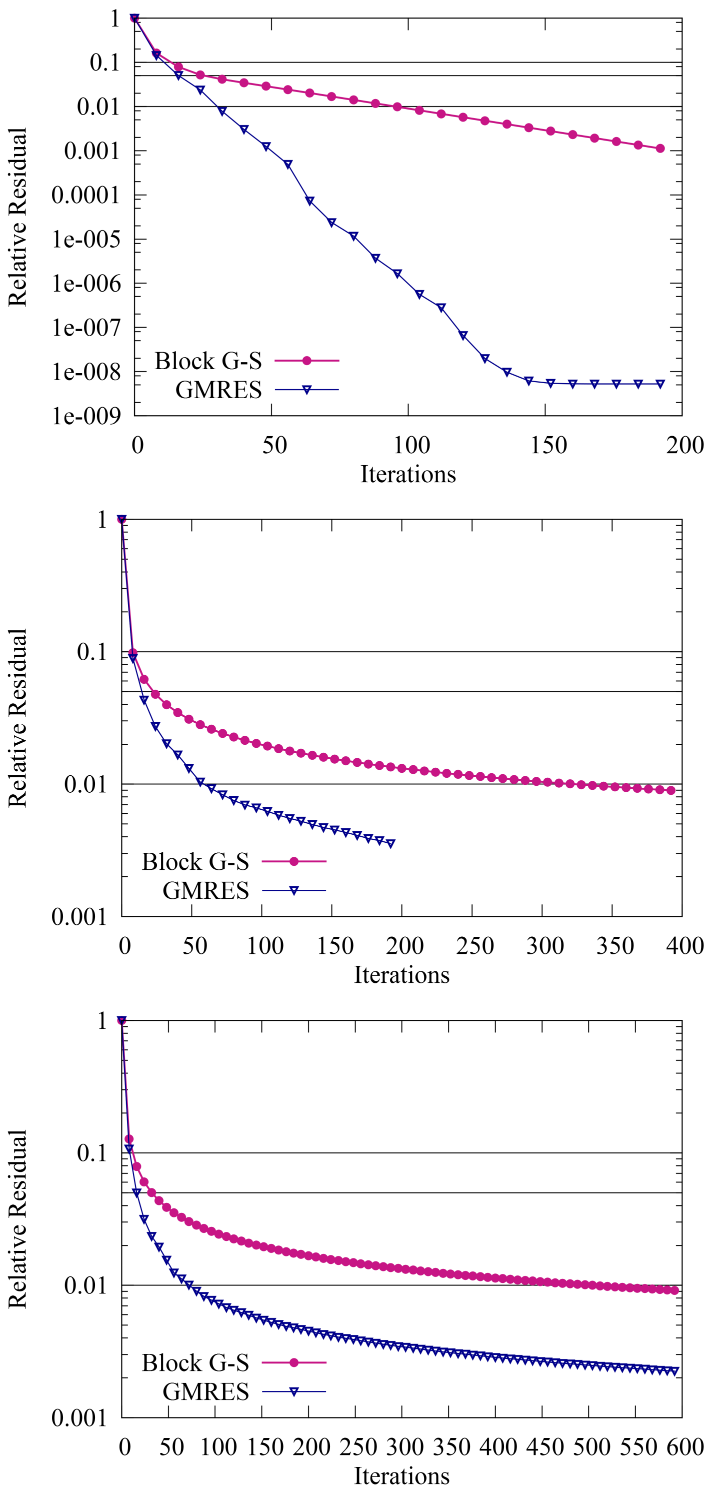


Figure 5.1: Convergence rates for a sample pressure equation extracted from a two-dimensional lid-driven cavity simulation, with 225 cells (left), 14.4k cells (middle) and 1m cells (right) – comparing GMRES to Block Gauss-Seidel. Horizontal lines represent 0.1, 0.05 and 0.01 relative convergence respectively. The relative residual is the residual of the system normalized to the residual at the initial condition.

computations [31][1]. Figure 5.2 illustrates this algorithm. Similar improvements have been invented for the conjugate gradient solver [61, 108]; both papers quoting massively-parallel computing architectures as the motivation for more scalable algorithms.

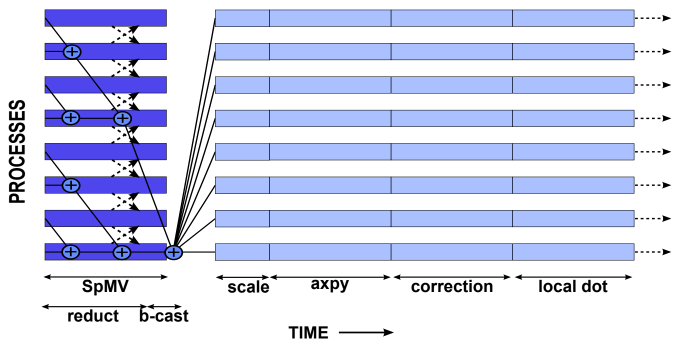


Figure 5.2: Illustrative parallel trace of a Pipelined GMRES iteration, showing the computation and communication patterns when utilizing 8 cores. Not to scale.

5.1.4 Algebraic Multigrid Methods

For elliptic equations (such as the pressure equation) algebraic multigrid methods such as ML [29] should be very powerful. Multigrid methods cover a broad category, with multiple formulations and many opportunities for fine-tuning. They are all based on the principle that multiple scales of the problem can be solved efficiently by solving coarse-grid approximations to the actual (fine) grid. The coarsest grid will have a much lower spectral radius than the finest, allowing low-frequency errors to be reduced quickly. Meanwhile, the fine grid solves high-frequency errors, and the results are combined. Since each grid is much easier to solve, 'smoothers' are used instead of complete solvers at each level. A typical smoother may just be one iteration of SOR or an ILU factorization, for example, although the coarsest grid is often solved directly. There are many variations of multigrid methods: different methods for coarse-grid construction; different methods for coarse-grid interpolation; various methods of communicating on coarse grids; and so on. All of these variations will have a large effect on scalability.

An ideal multigrid method can solve a linear equation-system in $O(1)$ iterations, with work-per-iteration increasing with $O(N)$. Thus the entire multigrid algorithm scales with $O(N)$. However, this ideal situation is rarely obtainable, and depends on the ability of the coarse-grid operators to reduce the maximum eigenvalue effectively [68, 103].

ML is a state-of-the-art smoothed-aggregation algebraic multi-grid method from Sandia's National Laboratories, and is one of the most commonly-used multigrid packages [29]. ML automatically creates coarse grids until a minimum size is reached, using a smoothed aggregation process. For the KVLCC2 test case, ML automatically creates six grids when running on one core, and four grids on 512 cores. Multigrid methods,

[1]'Pipelined' in this context is not related to instruction-level parallelization as introduced in section 2.2

including ML, can be used as standalone solvers or preconditioners for a Krylov Subspace method.

Further discussion on multigrid methods will be introduced in chapter 7.

5.1.5 Preconditioning

To improve the convergence rates of non-stationary solvers, preconditioning is often applied. The principal is to transform the coefficient matrix $\mathbf{A}$ into something less stiff, maintaining a system with the same solution, using a preconditioning matrix $\mathbf{P}$:

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{x} = \mathbf{P}^{-1}\mathbf{b} \tag{5.6}$$

The difficulty is in finding a matrix $\mathbf{P}$ which approximates $\mathbf{A}^{-1}$ and is easily invertable. At the extremes, if $\mathbf{P}$ is the identity matrix $\mathbf{I}$, finding $\mathbf{P}^{-1}$ is trivial, but the system is unchanged. At the other extreme, if $\mathbf{P} = \mathbf{A}^{-1}$, then the system is trivial to solve, but the calculation of $\mathbf{P}^{-1}$ is as difficult as the initial problem.

Finding a trade-off between cost of preconditioning and its effectiveness is difficult, and depends on the matrix system, the solver that will be used on the preconditioned system, and the number of iterations that are likely to be performed. The simplest preconditioner, the *Jacobi* preconditioner, takes $\mathbf{P} = \mathbf{D}$ (the diagonal of $\mathbf{M}$), and has the effect of normalizing the equation system. More complex methods, such as *Incomplete Factorization* (ILU) perform semi-complete direct methods to obtain $\mathbf{P} \approx \mathbf{A}^{-1}$.

Preconditioning is usually applied to all KSP solvers, since the simplest preconditioners are relatively cheap compared to a KSP iteration. Stationary methods rarely use preconditioners, since the preconditioners would mostly be more expensive than the solver itself.

In most KSP methods, the residual is computed as part of the solution process, thus additional communications are not required. When preconditioning a linear system, it becomes difficult to compute accurate residuals. With left preconditioning, described here, these solvers base their convergence on the preconditioned residual $\mathbf{P^{-1}r} = \mathbf{P^{-1}}(\mathbf{b}-\mathbf{Ax})$ which is *not* proportional to the true residual. If the preconditioner does not treat all error frequencies equally, and particularly if the matrix is close to singular (spectral radius close to 1) then an ILCT of 0.01 in the preconditioned residual may be equivalent to a much higher ILCT in the *true* residual.

For the Block Jacobi preconditioner used in the previous chapter, experiments were conducted to determine the magnitude of this effect. An ILCT of 0.1 in the preconditioned residual was equivalent to just 0.6 in the true residual, but was less obvious with other preconditioners. This effect had previously been unnoticed in ReFRESCO, and caused issues in early experiments. This issue is rarely documented in literature, and probably affects other CFD packages.

To circumvent this problem, a true residual can be computed at the end of each iteration, but this requires additional global communications. Instead, one can solve the right-preconditioned system:

$$\mathbf{AP}^{-1}\mathbf{Px} = \mathbf{b}$$

via the solution of $\mathbf{AP}^{-1}\mathbf{y} = \mathbf{b}$ and $\mathbf{Px} = \mathbf{b}$. This leaves the true residual intact. For GMRES, another solution is to switch to Flexible GMRES (FGMRES) – a modified version of GMRES which preserves the residual in its true state.

ReFRESCO has switched to using right preconditioning instead of left preconditioning, in order to obtain a more consistent and predictable solution to the linear system. Unless specified, right preconditioning is used in this thesis – but comparisons to a left-preconditioned FGMRES are also performed. Early experiments, such as in Hawkes et al. [36], suffered because of left-preconditioned residuals.

5.2 Scalability Experiments

Thus far, scalability experiments have used a basic GMRES solver with a Block Jacobi (right-)preconditioner. More modern solvers or preconditioners could provide very different scalability characteristics. For example, a powerful preconditioner could reduce the number of KSP iterations (and global communications) required; but may be unscalable in itself due to communication or high set-up costs. CFD is unique in that a solution to the linear system is only solved to a relatively lax ILCT, since the solution to the linear system is a small part of a non-linear system. Compared to applications which require machine accuracy of linear systems, CFD is very sensitive to start-up (initialization of linear solvers, memory allocation, etc.) costs which may rule more advanced solvers or preconditioners. Indeed, it may be that the simpler preconditioners, such as Block Jacobi, provide the best scaling. In this section, the scalability of various linear solvers will be tested. Following this, a number of preconditioning techniques will be investigated – including block preconditioning techniques, multigrid preconditioners and simple smoothers.

5.2.1 Alternative Solvers

The scalability of various solvers has been tested using the aforementioned test cases (KVLCC2 and LDCF). The same parameters as in chapter 4 are used. Instead of showing scalability for all profiled routines, the scalability plots are adjusted to show just the solve routines for the pressure equation. Note that the embedded core-hours plots in the following scalability plots utilize a logarithmic scale due to large differences in performance.

The scalability of GMRES, Pipelined-GMRES, Flexible-GMRES, BCGS, Improved-BCGS, SOR, and a multigrid method (ML) are compared in figure 5.3. The four Krylov Subspace methods have been preconditioned with a Block Jacobi preconditioner. Right preconditioning is used for all solvers except FGMRES, which uses left preconditioning but still provides a true residual. The SOR solver is configured to compute a residual every 100 iterations, and $\omega = 1$ (equivalent to Gauss-Seidel). An ILCT of 0.1 and 0.01

is tested. All of the solvers and preconditioners presented are provided by PETSc [6], except ML – which is provided by Trilinos [29, 41] via a PETSc interface. Some KSP methods were not performed on the LDCF test case.

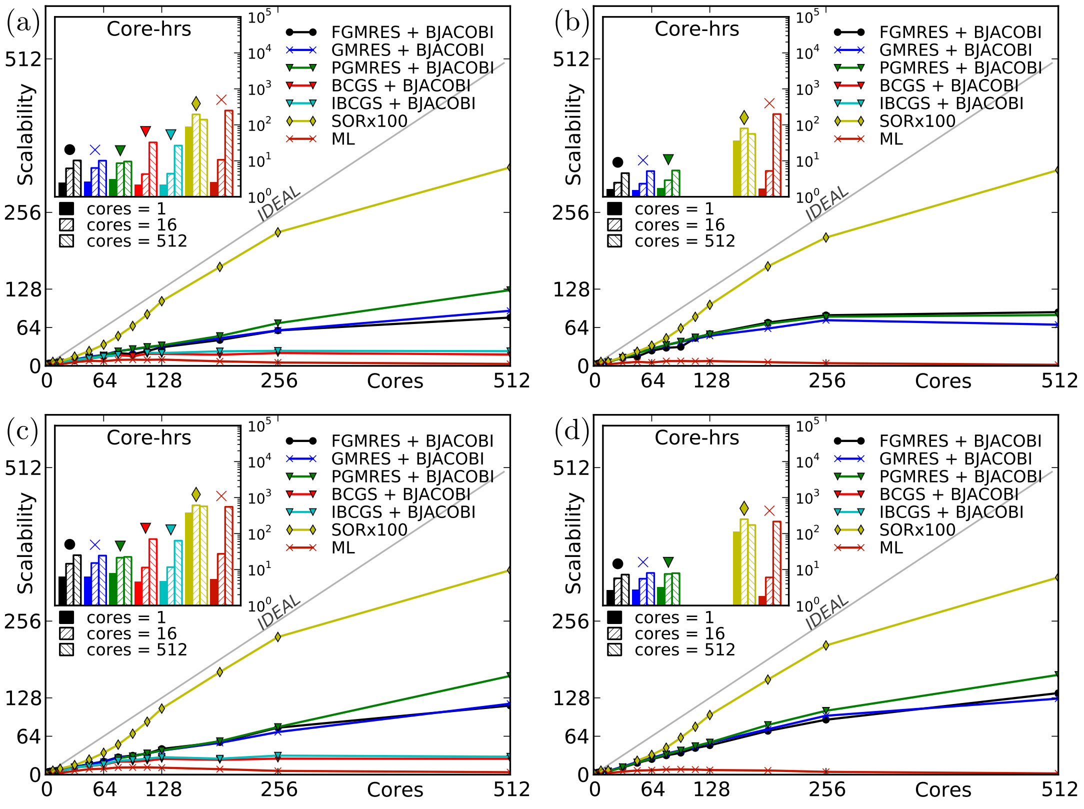


Figure 5.3: Scalability of the solve routines in the pressure equation, using the KVLCC2 test case (left) and LDCF test case (right) with inner-loop convergence tolerance set to 0.1 (top) and 0.01 (bottom). Note the exponential scale of the inset core-hours chart.

GMRES performs as previously noted, with poor performance at the intra-nodal level due to memory-bandwidth contention and poor performance on a large number of cores due to global communication patterns. FGMRES exhibits similar scalability to GMRES. PGMRES scales worse in the memory-bandwidth zone than either GMRES or FGMRES, but the gradient of the scalability factor, dS/dC, between 256 and 512 cores is approximately double that of GMRES – consistent with the algorithmic improvement (half the number of global communications). Overall, the wall-time gains from PGMRES on 512 cores are minimal.

BCGS gives strong numerical performance in serial operation and similar memory-limited scaling at the intra-nodal level. BCGS requires four global communications per iteration (instead of two), thus inter-nodal scaling suffers. IBCGS improves on these results slightly.

As expected, SOR performs much worse than the KSP methods, but has far superior scalability. SOR is limited by memory bandwidth briefly, but quickly recovers this as the simulation fits into cache. The SOR algorithm still uses one global communication pattern to compute a residual every 100 iterations; though this did not appear to be significant when in-depth profiling results were examined. Time spent in MPI_Recv was dominant, implying that implicit synchronization is the bottleneck to scalability where $C > 256$.

ML is very strong in serial operation, but quickly loses ground to other solvers due to poor scalability. It is not completely clear what causes this. Profiling of the solver reveals a lot of time spent in local MPI communications (`MPI_Recv`), which could be caused by the smoothers used on each level; particularly the very coarse levels. However, this could also be caused during the aggregation/set-up of the coarse grids. The convoluted way in which ML is exposed to ReFRESCO via PETSc makes the profiling much more difficult. Some other permutations of ML are tested later in this chapter, when applying it as a preconditioner to FGMRES.

5.2.2 Block Preconditioners

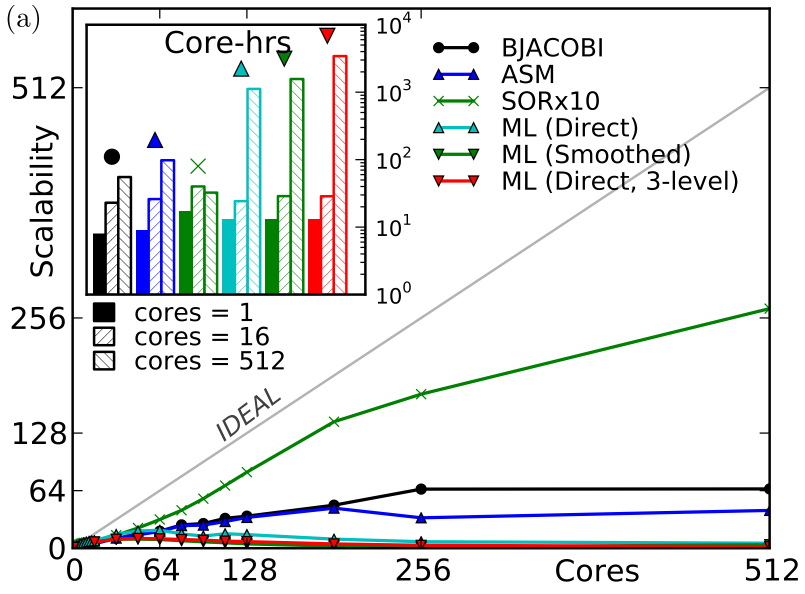
ReFRESCO has access to a large range of preconditioners through its use of PETSc, many of which have been tested as follows, and collocated into a single figure (5.4). Left-preconditioned FGMRES was used for these tests. Its flexibility is specifically designed to allow complex preconditioning, such as multigrid preconditioners, which change $\mathbf{P}$ every iteration. Only the KVLCC2 test case was used to test the majority of these preconditioners, due to the amount of simulations required to perform these experiments. Many of these experiments were repeated using right-preconditioned GMRES and PGMRES, with no noticeable differences (not shown).

The Block Jacobi algorithm used thus far implements a block-wise Incomplete LU (ILU) factorization with zero-level fill, and sets a high benchmark for other preconditioners. ILU(0) is performed on each MPI process's local portion of the matrix, leading to an interesting problem: convergence deteriorates as the number of cores increases, as the local portion becomes less significant to the global solution. An Additive Schwarz Method (ASM) was tested, with the same block-wise ILU(0) solver. ASM is similar to Block Jacobi, but allows communication between neighbouring blocks to augment the process and prevent this degradation of convergence. The results are shown in figure 5.4. The differences between Block Jacobi and ASM were small, with ASM fairing worse overall. ILU(0), ILU(1) and ILU(2) were also tested as preconditioners in their own right, with poor results in all regards (not shown).

5.2.3 Multigrid Preconditioners

Multigrid methods such as ML can also be used to precondition the equation system. The results, also shown in figure 5.4, show that ML is highly capable at the intra-nodal level – just as when it is used as a standalone solver. It exhibits strong serial performance and moderate scaling to 16 cores – better than Block Jacobi. Unfortunately it rapidly breaks down beyond 64 cores. It was suspected that this was due to the direct solver used on the coarsest grid, but replacing it with a smoother (5 iterations of SOR) resulted in worse scalability. Another recommendation is to restrict the number of levels created by ML, thus reducing expensive start-up costs. Restricting ML to three levels worsened the scalability; and two levels (not shown) gave similar results. It is expected that less

(a)



(b)

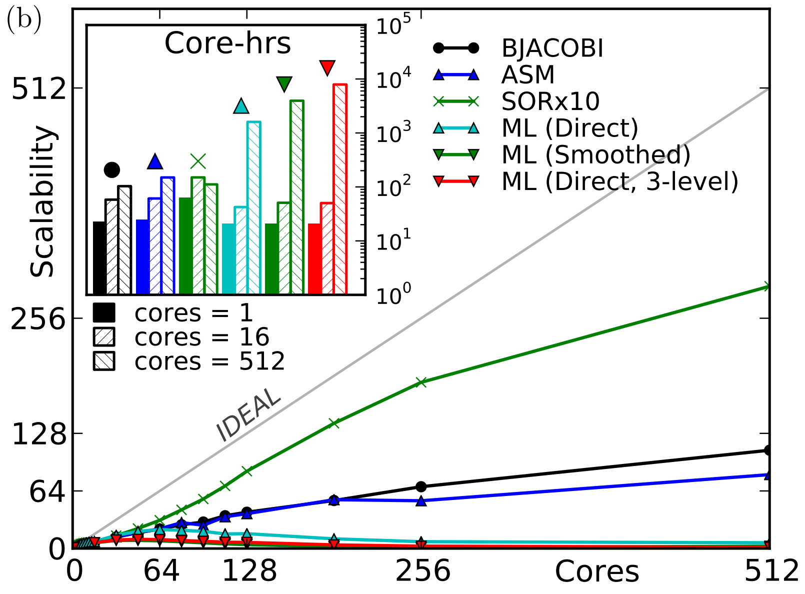


Figure 5.4: Scalability of the solve routines in the pressure equation, using the KVLCC2 test case with inner-loop convergence tolerance set to (a) 0.1 and (b) 0.01. The results compare different preconditioners including Block Jacobi, Additive Schwarz (ASM), SOR smoothing (SORx10) and a multi-grid method (ML). FGMRES is used as the solver to allow more flexible preconditioning. Note the exponential scale of the inset core-hours chart.

sophisticated multigrid methods (such as non-smoothed aggregation) may provide better results for CFD, since the start-up cost of creating coarse-grid operators is lower; but coarse-grid smoothing is likely to remain a bottleneck. Clearly a much deeper study of multi-grid methods is required, ML certainly cannot be used as a black-box for scalable CFD – neither as a preconditioner nor as a standalone solver.

5.2.4 Smoothing Preconditioners

Finally, it is worth considering a much simpler preconditioner than even Block Jacobi. Instead of preconditioning in the classic sense, a *smoother* can be used before the main solver. Ten iterations of SOR as a smoother was optimal (compared to 1, 100 or 1000), with $\omega = 1$, such that SOR becomes a Block Gauss Seidel method. Although it performed worse than the Block Jacobi preconditioner in serial operation, where Block Jacobi has good convergence, it was able to catch up due to memory caching, thus providing excellent scalability. Since a fixed number of smoother iterations were performed, residual computation in the SOR algorithm was unnecessary. This combination has minimal set-up costs, since SOR requires no additional memory or pre-computation, so could be a viable option for scalable CFD. The scalability is slightly worse than the standalone SOR solver, due to poor scaling of the overarching FGMRES solver. Furthermore, the numerical performance is not good compared to other preconditioners, as the smoother cannot reduce low-frequency errors effectively – it is only competitive on a large number of cores. Overall, this FGMRES+SOR combination appears to be a good middle-ground between the scalability or SOR, combined with the numerical performance of FGMRES.

5.3 Summary & Conclusions

This section has looked at the theory and performance of various linear solver and preconditioners. The most promising result was FGMRES with 10 iterations of SOR as a preconditioner. Although it was the slowest preconditioner for serial computation, superior scaling meant that it was often the best performer at $C = 512$. The initial scalability study from chapter 4 is re-illustrated using this configuration in figure 5.5. Inter-nodal scaling is significantly improved (compared to figure 4.4.a), which is encouraging, but parallel efficiency is still poor and global communications are still limiting. Overall wall-time on 512 cores has only been improved by approximately 10%, and with stricter convergence tolerances these gains are likely to be lost due to the poor numerical properties of SOR.

Overall, no solver or preconditioner has significantly and reliably solved the strong scalability problem. In terms of scalability, the most interesting solutions came from stationary solvers and smoothers; although these only became viable on a large number of cores and are likely to lose ground when a stricter ILCT is required, or when N increases, due to their spectral-radius-limited convergence rates. State-of-the-art scalability improvements to existing solvers, such as PGMRES, provided minimal improvements – it

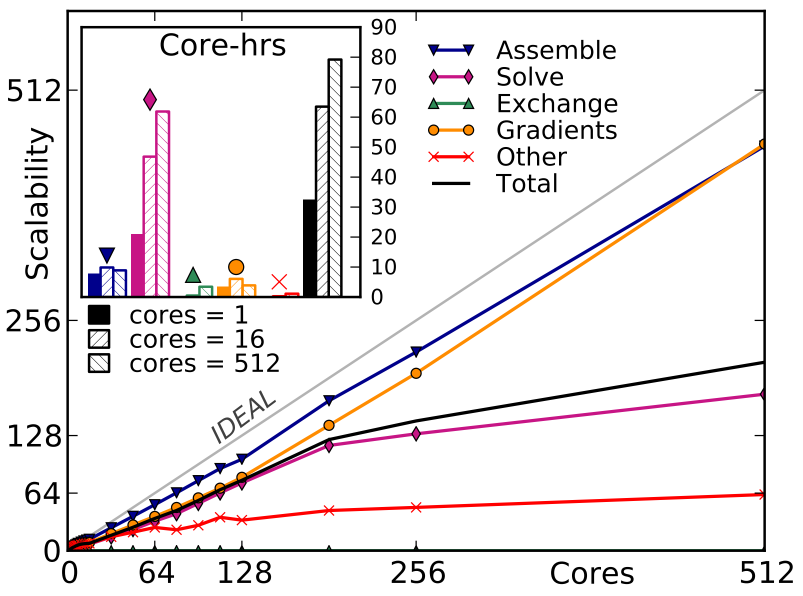


Figure 5.5: Scalability of the code for the KVLCC2 test case, and the profiled routines within, as the number of cores increases. These results used FGMRES with 10 iterations of SOR as a smoother, with an inner-loop relative convergence tolerance of 0.1. Similar results were found for the LDCF test case.

appears that solvers relying upon global communications are fundamentally unscalable. Multigrid methods are certainly an area which warrants further investigation, since their numerical performance in serial operation is very good; and there are many permutations of the multigrid algorithm which may offer better scalability, by completely avoiding global communications. However, coarse-grid communications appear to cause scalability issues, especially as the cells-per-core ratio becomes small.

In general, the state-of-the-art focuses on improving the scalability of numerically-powerful solvers such as GMRES [31, 61, 108] – but this has not solved the scalability problem. In this PhD, an alternative strategy is employed by building on the results of the simple SOR solvers. A more performant version of SOR is investigated in the following chapter, using the theory of *chaotic relaxations* to remove the implicit synchronization caused by neighbour-to-neighbour communications. To begin, these solvers will be used as standalone solvers, thus limited to $O(N^3)$ performance. In chapter 7 this *chaotic solver* will be shaped into a chaotic multigrid method, where the improved efficiency of these communications will hopefully remove the scalability problems associated with coarse-grid communications of classical multigrid methods.

Chapter 6

Chaotic Linear Equation System Solvers

In the previous section, the performance of stationary solvers was shown in terms of performance and scalability. The main bottleneck to scalability of these simple solvers is the implicit synchronization imposed at the end of each iteration. This motivates research into *chaotic methods*, which use the theory of *chaotic relaxation* [16] to desynchronize iterations and completely remove synchronization. In this chapter, these chaotic methods are investigated to create a standalone *chaotic solver*, which may be able to compete directly with preconditioned GMRES. As will be shown, the chaotic solver does indeed outperform KSP methods on a large number of cores. However, due to the intrinsic nature of stationary solvers – with performance proportional to $O(N^3)$, chaotic solvers are not a sustainable solution for scalable CFD. In following chapters, however, these chaotic principles will be applied to a chaotic multigrid method.

This chapter begins by discussing the theory and applicability of chaotic methods to CFD, and then follows the implementation, verification and scalability-testing of a chaotic solver. As part of this PhD, an open-source library, *Chaos*[1] has been published – which includes the implementation of the chaotic solver[2].

6.1 Theory of Chaotic Relaxations

Consider the relaxation of each equation in the Jacobi method, as presented in section 5.1.1:

$$x_i^k = \left(-\sum_{\substack{j=1 \\ j\neq i}}^{N} a_{ij}x_j^{k-1} + b_i \right) / a_{ii}, \quad i = 1, \ldots, N. \tag{6.1}$$

where a, x and b are the individual components of $\mathbf{A}$, $\mathbf{x}$ and $\mathbf{b}$ respectively. At the end of each iteration the new values of $\mathbf{x}$ must be globally communicated before the next iteration can begin. Numerically, improvements to this scheme can be made by

[1]`https://bitbucket.org/jamesnhawkes/chaos`

[2]The chaotic solver implementation can be found in `Solver::ChaoticSolve()` (`src/solver_chaotic.cpp`) at the time of writing.

using values from the current iteration (k) as they are available, thus creating the Gauss-Seidel method. It is also possible to prove the convergence of the parallel block Gauss-Seidel methods, where an arbitrary selection of variables from iterations k and $k-1$ are used, based on the partitioning of the domain. All of these methods have an implicit synchronization point, because no two processes can become more than a single iteration out of sync – no process may proceed to iteration k until iteration $k-1$ is complete on all of its neighbours.

In 1969, Chazan & Miranker [16] proposed the concept of *Chaotic Relaxation* whereby several processes (distributed-memory processes or shared-memory threads) never synchronize. Instead, the processes freely pull values of off-diagonal x_j from memory whenever they are required, and push new values for the diagonal x_i whenever they have been relaxed. In this way, each relaxation uses the values of $\mathbf{x}$ from the latest iteration that is available – this could be several iterations behind the current relaxation iteration ($s = 1, \ldots$,n), or even ahead of it ($s < 0$):

$$x_i^k = \left(-\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij} x_j^{k-s} + b_i \right) / a_{ii}, \quad s < s_{max}, \quad i = 1, \ldots, N. \tag{6.2}$$

The order in which the n equations are relaxed is completely arbitrary. With this scheme, processes never need to wait for each other. Although communication must still occur, it can be entirely asynchronous, thereby making efficient use of memory bandwidth, inter-nodal communications and computational resources. Processes may even iterate multiple times on the same data if necessary, making the best use of the available hardware. Whilst this method is based on the stationary Jacobi method, s can vary between iterations implying that chaotic methods are actually non-stationary, although numerically this makes little difference.

Chazan & Miranker [16] proved that this iterative scheme will converge for any real Jacobi iteration matrix if $\rho(|\mathbf{M}|) < 1$ so long as s_{max} is bounded. $|\mathbf{M}|$ represents the Jacobi iteration matrix where all the components have been replaced with their absolute values. The implications of s_{max} being bounded is simply that if two relaxations take different amounts of time, they cannot be left completely independent indefinitely, such that s could potentially become infinite. Baudet [10] went on to prove that $\rho(|\mathbf{M}|) < 1$ is a necessary condition for convergence for any $s_{max} \leq k$. Bahi [5] further showed that $\rho(|\mathbf{M}|) = 1$ is also valid, if $\mathbf{M}$ is singular and s_{max} is bounded. In this situation, the error associated with the maximum eigenvalue (1) is not reduced, but the solution can still be solved up to a constant value, with a rate based on the second-highest eigenvalue of $|\mathbf{M}|$. This occurs in CFD simulations where the pressure is not bounded by any Dirichlet boundary conditions.

The extent to which chaotic relaxation is exploited varies significantly in the literature, and most commonly the exact implementation is not discussed. In Anzt et al. [2] the theory is used to create a 'block-asynchronous' Jacobi solver for GPUs, where a fixed number of local iterations are performed between communications. These solvers are often referred to as *asynchronous* methods, such that *async(5)* refers to a Jacobi method with communications every 5^{th} iteration. This reduces implicit synchronization and shows good results compared to stationary methods, demonstrating that relaxations on 'old' data ($s > 1$) are not wasted. However, this method could also needlessly throttle communications, thus reducing convergence rates if communication hardware is not saturated. Anzt et al. focuses on GPU architectures, where this may not be such a problem.

In this thesis, chaotic relaxation theory is leveraged further. By separating communications and computations into shared-memory threads, the fixed asynchronicity can be removed and the respective hardware can be saturated, leading to increased efficiency and numerical performance. In this truly chaotic method, computation threads never need to wait for communications to be complete and communication hardware is never wasted waiting for computations. Furthermore, a multi-threaded model can exploit access to shared memory, inserting updated values (from communications) directly into the active relaxations, which should give numerical advantages. The targeted architecture for this *chaotic solver* is a hybrid-parallel (MPI + OpenMP) CPU environment; but it would be possible to extend these chaotic methods to heterogeneous environments with relative ease.

The convergence criteria for chaotic relaxations, $\rho(|\mathbf{M}|) < 1$, is stricter than that of standard stationary methods, which only require $\rho(\mathbf{M}) < 1$, and stricter still than the oft-used Krylov methods (which have no such requirements). The following section aims to determine if a range of standard CFD test cases will produce matrices that satisfy the criteria, thus testing the applicability of chaotic relaxations to CFD.

6.2 Determining Convergence Criteria for CFD

All the equation-systems in CFD are diagonally dominant by design, which is a sufficient condition for convergence of standard stationary methods – since diagonal dominance implies the necessary condition $\rho(\mathbf{M}) \leq 1$. It also implies that $\rho(|\mathbf{M}|) \leq 1$ is satisfied, since the diagonal dominance is computed, on each row, as:

$$|M_{ii}| - \sum_{j \neq i} |M_{ij}| \geq 0, \quad i = 1, ..., n.$$

In this section, a number of linear equation systems are extracted from typical test cases, and $\rho(|\mathbf{M}|)$ is calculated to demonstrate this diagaonal dominance and adherence to the convergence criteria. This also provides a good opportunity to investigate the matrices produced in the SIMPLE algorithm and gain more insight into the linear systems. Some additional transport equations, which may affect $\mathbf{M}$ (by modifying the right-hand-side of the equation system) can be tested, along with some simulations with more complex flow features. The following test cases are chosen:

- **2D Lid-Driven Cavity Flow** (LDCF) on a number of structured grids (225, 14.4k, 1m elements). No turbulence models. See Klaij & Vuik [52].

- **NACA0015** hydrofoil (15° angle-of-attack) with a two-dimensional multi-block structured grid (28k elements) and a two-eqn. κ-ω SST turbulence model [63]. See Rijpkema [79].
- **NACA0015(C)**, as above, but with a Sauer-modified cavitation model. See [45].
- **KVLCC2** with a three-dimensional multi-block structured grid (317k elements) and a hexahedral unstructured grid (**U**) with hanging nodes (167k elements). Using a κ-ω SST turbulence. See chapter 4.
- **Dambreak**, a homogeneous two-phase (air/water), three-dimensional problem with a simple structured grid (16k cells) and a volume-fraction equation. No turbulence model. See [99].
- **Cylinder**, a low Reynold's number, unsteady (10 non-linear loops per time-step) simulation with a structured grid (4.3k elements) and a poor initial flow estimation (designed to create ill-conditioned matrices). No turbulence model. See [80].

The matrices were extracted at outer loops 1, 5, 10, 50 and 100. The momentum equations (in x, y and z) are identical, due to the way in which they are assembled. The size of mesh that could be extracted and examined was limited because the matrices had to be inverted directly, using Gaussian-Elimination.

A C++ tool was developed to perform analysis on these matrices. For each matrix, it is possible to plot the connectivity graph, following the methods of Hu [47] – this gives a qualitative, visual insight into the sparse matrices by graphically connecting the elements of the matrix. The sparsity pattern of the matrix $\mathbf{A}$ may also be plotted directly in order to view the matrix structure.

The largest eigenvalues of $|\mathbf{M}|$ can be found using ARPACK [58] routines, and plotted in an Argand diagram – for $\rho(|\mathbf{M}|) \leq 1$ all eigenvalues must lie within a unit circle.

Assuming a satisfactory spectral radius, the *condition number* (the ratio of the largest to smallest eigenvalue) of the original matrix $\mathbf{A}$ can be used to assess the sensitivity of the equation system to errors in the input. For example, in the solution to the pressure-equation, approximations (geometric, interpolation, rounding, etc.) are transferred to $\mathbf{A}$ and $\mathbf{b}$. When the solution to the linear system is found, these errors will have been amplified by the condition number into the solution vector $\mathbf{x}$. The condition number is computed using a 1-norm condition estimator [35].

Figure 6.1 shows a qualitative view of the simple 2D LDCF, the more complex 2D hydrofoil, and the 3D KVLCC2 case at the 5^{th} outer loop. The connectivity graphs show resemblance to the underlying geometry and mesh structure – for example, the NACA0015 connectivity resembles the O-grid from which it arose.

The sparsity patterns are closely related to the structure of the grid and the cell-numbering – the more complex KVLCC2 case is highly irregular compared to the Cartesian, structured LDCF grid. It should be noted that the matrices were extracted from single-process simulations, so the matrices have not been subject to domain decomposition. It would be interesting to extract parallel matrices and observe their communication patterns, but this was beyond the scope of this analysis.

Table 6.1 shows the quantitative results from all the test cases, taking the maximum values of all the extracted outer loops. The condition number appears to have some correlation with spectral radius and some are worryingly high. In all cases, $\rho(|\mathbf{M}|) \leq 1$, meeting the requirements for chaotic solvers as expected.

The pressure equation was virtually singular for the LDCF and Dambreak case, where only Neumann boundary conditions are applied – specifying a null space may help to reduce the condition number in these cases. In all the other cases, a Dirichlet condition on the outflow reduced the spectral radius, although it was still close to one; and the condition number also dropped considerably. For the unsteady cylinder, the spectral radius was also very high – these cases both feature complex flows with poorly-initialized flows and coarse meshes; this large discrepancy and poor resolution could be the reason for the the near-singular matrices.

Additional matrices for the KVLCC2 case (up to 2.67m elements) and a 21.7m-element INSEAN E779A propeller (with sliding interface) were tested, but could not be fully post-processed due to memory requirements. However, the spectral radius of the pressure equation was computed at a single iteration for these cases, and was shown to satisfy $\rho(|\mathbf{M}|) \leq 1$. It took 2.5 days and 232 GB of memory to compute the maximum eigenvalue of a single matrix for the INSEAN propeller. Optimizations may be possible to check other matrices, but the test cases appear conclusive.

In all-but-one case (the ill-conditioned cylinder at the first time-step), the momentum, turbulence, cavitation and volume-fraction equations all exhibited much lower spectral radii. Thus demonstrating that these transport equations do not contain low-frequency errors, as expected.

Table 6.1: Quantitative results for a range of test matrices, showing condition number and spectral radius for momentum-, pressure-, turbulence- and free-surface-/cavitation-equations. Note that the spectral radius was always less than or equal to one, within double-precision floating-point tolerances.

	Mom.		Pres.		Turb. 1		Turb. 2		V.F./Cav.	
	cond($\mathbf{A}$)	$\rho(\lvert\mathbf{M}\rvert)$	cond($\mathbf{A}$)	$\rho(\lvert\mathbf{M}\rvert)$	cond($\mathbf{A}$)	$\rho(\lvert\mathbf{M}\rvert)$	cond($\mathbf{A}$)	$\rho(\lvert\mathbf{M}\rvert)$	cond($\mathbf{A}$)	$\rho(\lvert\mathbf{M}\rvert)$
LDCF-225	210	0.955	9.4·e17	1.000						
LDCF-14.4k	7	0.500	2.1·e18	1.000						
LDCF-1m	4	0.500	2.9·e19	1.000						
NACA0015	9.0·e3	0.884	6.53·e6	0.999	4.5·e3	0.849	4.8·e3	0.851		
NACA0015(C)	6.3·e3	0.734	4.67·e6	0.999	5.1·e3	0.688	5.2·e3	0.690	2.3·e8	0.240
KVLCC2	2.6·e6	0.845	2.63·e8	0.999	1.2·e7	0.588	1.2·e7	0.594		
KVLCC2 (unst.)	5.2·e6	0.810	5.4·e7	0.999	8.3·e6	0.726	7.7·e6	0.728		
Dambreak	105	0.048	5.6·e18	1.000					7	0.048
Cylinder (t=1)	111	0.999	1.1·e6	0.999						
Cylinder (t=2–10)	4.4·e2	0.520	1.1·e6	0.999						

6.3 *Chaos*: An Open-Source Library for Chaotic Methods

The chaotic methods discussed in this paper have been written as a standalone library, *Chaos*[3]. The *Chaos* library is available under an open-source MIT license, in the hope

[3]`https://bitbucket.org/jamesnhawkes/chaos`

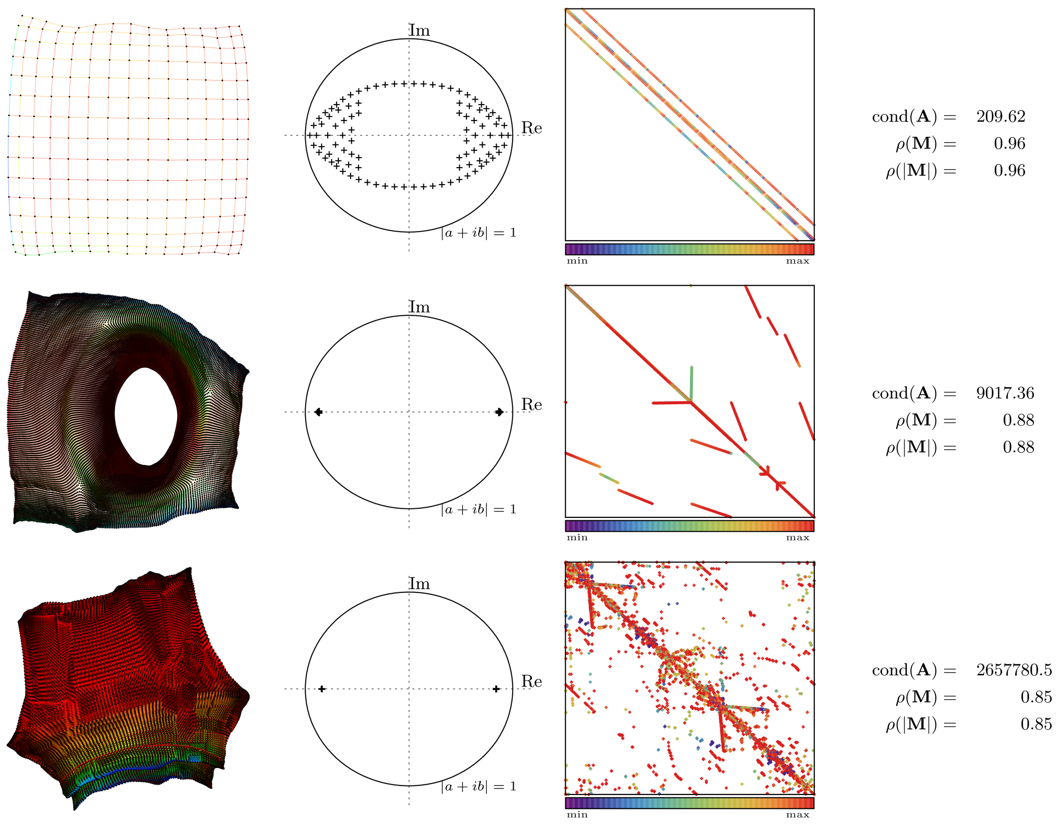


Figure 6.1: Connectivity, 100 largest eigenvalues, sparsity and statistics for the momentum equation (outer loop 5). [Top] LDCF-225, [Middle] NACA0015-28k and [Bottom] KVLCC2-317k.

that the results herein can be reproduced, developed and applied to other scientific codes and disciplines. The library is written in C++ and can be bound to Fortran and C easily. Bindings are also created for Python and Ruby automatically, using SWIG [93]. Many other languages could also be bound easily. *Chaos* is a hybrid-parallel library, using MPI and OpenMP, and has been tested under multiple compilers, multiple MPI implementations and multiple operating systems.

The interface is similar to other linear-solver toolkits, such as PETSc [6] and Trilinos [41]; it should be easy for codes which use these libraries to interface with *Chaos*. The main difficulty is in connecting MPI-only codes (such as ReFRESCO) with the hybrid MPI+OpenMP scheme used in *Chaos*. Although most of *Chaos* can be run in MPI-only mode, the chaotic methods developed here rely on the hybrid, multi-threading layer of parallelization. For now, ReFRESCO is left in MPI-only mode, but with most of the CFD code running on $\frac{1}{8}^{th}$ of the available hardware. A long-term solution would be to add OpenMP parellelization to ReFRESCO, which may have benefits as discussed in section 3.2. For true mixing of hybrid and non-hybrid codes, shared-memory paradigms of MPI (such as one-sided communication) could be considered.

Currently, the *Chaos* library includes over 4500 lines of code, and several basic examples/tests. This library, with accompanying publication focused on chaotic solvers and chaotic multigrid methods (Hawkes et al. [40]), represents the largest deliverable from this PhD – and a large, novel contribution to the fields of CFD and numerical analysis.

One can easily make the argument that all of the solvers described in this thesis could have been built within the framework of PETSc. Indeed, the first iteration of chaotic solver was written in PETSc. Unfortunately, PETSc recently rescinded its support for hybrid parallelization, and it thus became infeasible to maintain these solvers within PETSc. Regaining control over much of the matrix structure and communication patterns also provided advantages when implementing chaotic methods. Most of the interfaces in PETSc are designed to be abstracted and safe; when writing chaotic methods it is common to hack these methods to allow deliberate data races and 'unsafe' asynchronism.

6.4 Implementation

Consider the matrix $\mathbf{A}$, partitioned contiguously across multiple MPI domains, such that each process owns a block of n rows of length N. It also owns the corresponding portion of the vectors $\mathbf{x}$ and $\mathbf{b}$. To perform a chaotic relaxation on this block, as in equation 6.2, the off-diagonal elements of the matrix are multiplied by the solution vector. Where these off-diagonal elements correspond to the local portions of $\mathbf{x}$ this is trivial. For non-local elements, the value of $\mathbf{x}_j$ must be obtained from another process via MPI communications.

For the purposes of implementation it is helpful to store the partitioned matrix as two compressed-row-storage (CRS) matrices, $\mathbf{A_A}$ and $\mathbf{A_B}$, as shown in figure 6.2. $\mathbf{A_A}$ is an n-by-n square matrix (containing the diagonal) which can be directly multiplied by $\mathbf{x_A}$, the local portion of $\mathbf{x}$. $\mathbf{A_B}$ contains all the remaining off-diagonal elements, which must be multiplied by off-process elements $\mathbf{x_B}$. The structure of $\mathbf{A_B}$ and $\mathbf{x_B}$ are re-arranged so that $\mathbf{x_B}$ becomes a local, sparse representation of the parallel vector.

The process of updating this buffer is known as vector-scattering. Each process must pack (only) the required local values of $\mathbf{x_A}$ into an a buffer, which is sent to an neighbouring process via an MPI message, and inserted directly into $\mathbf{x_B}$.

In a classical Jacobi solver, it is common to perform part of the relaxation with $\mathbf{A_A}$ and $\mathbf{x_A}$ whilst the scatter is being performed, and complete the iteration using $\mathbf{A_B}$ and $\mathbf{x_B}$ when the communication is complete. In asynchronous methods, vector-scattering can be overlapped over a fixed number of iterations and old values of $\mathbf{x_B}$ are used in the meantime.

Here, we consider a truly chaotic solver in which scattering and relaxing occurs simultaneously on independent threads, and the relaxation uses the most up-to-date values of $\mathbf{x_B}$ that are available. To achieve this, a hybrid-parallel scheme is used. The matrix is still partitioned into MPI sub-domains, but within each MPI process several shared-memory OpenMP threads operate. It is commonly recommended to run such a system with each MPI process occupying a single socket (or NUMA-node) of the supercomputer so that all threads are using the same memory channels (avoiding 'Non-Uniform Memory Access' problems). Each OpenMP thread is then given one physical core. For example, a problem previously using 64 MPI processes may be replaced by

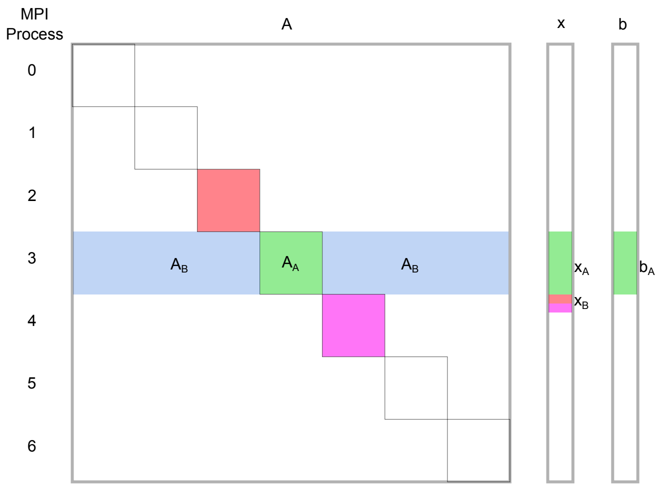


Figure 6.2: An illustration of the MPI parallel partitioning scheme for the matrix $\mathbf{A}$, and vectors $\mathbf{x}$ and $\mathbf{b}$. Note that the column indices of $\mathbf{A_B}$ are altered to point to the correct location in $\mathbf{x_b}$, the vector-scatter buffer.

8 MPI processes, each running 8 threads. Actually distributing MPI processes and threads correctly can be quite challenging. In the case of Iridis4, job host-files do not correctly describe the physical layout of the system, and manual intervention is required to correctly distribute MPI processes and set OpenMP thread affinity.

On each MPI process, one thread is designated as the *communications* thread and the remainder are assigned to *computation* (and perform the relaxations). This task-based thread assignment has been shown to provide benefits, even for synchronous methods, since the computation threads are often limited by memory bandwidth anyway [56].

The computation threads have shared-memory access to all of $\mathbf{A_A}$, $\mathbf{A_B}$, $\mathbf{x_A}$ and $\mathbf{x_B}$ and continuously perform relaxations on this data. Each thread takes a contiguous portion of the n rows and relaxes them in order (using equation 6.2), and repeats until the stop criteria are reached. It is not necessary to synchronize the computation threads, so each thread may perform a different number of iterations. Values of $\mathbf{x_A}$ are pushed and pulled from local memory without synchronization or mutexing. Values of $\mathbf{x_B}$ are pulled from local memory too, whilst the communication thread updates them.

The role of the communication thread is to continuously perform vector-scatter operations between MPI processes. It is responsible for collecting local values $\mathbf{x}_A$ and packaging them into MPI messages – once again, no synchronization or mutexing is required and these values are pulled directly from the active memory of the computation threads. On receiving a message, the incoming data is pushed directly into $\mathbf{x}_B$, immediately making it available for relaxations. In this implementation, a vector-scatter must be completed by the communication thread on all processes, before the next can begin. As such, every MPI process will perform the same number of communication iterations and will experience implicit synchronization of communication threads. This synchronization of communication threads does not affect the computation threads. A more chaotic scheme could be implemented, allowing pairs of processes to communicate faster than other pairs if the hardware allowed it. One-sided MPI communications would be the ideal method to pursue this, but this was not deemed necessary at this stage.

This chaotic solver deliberately exploits race conditions to achieve a high level of asynchronicity. Although no two threads write to the same location in memory simultaneously, it is quite possible that threads will try to read and write the same value at the same time. As long as the data type is atomic, this causes no issues. It is not transparent how the compiler or memory controller deals with cache-coherency when encountering these data races, but this also caused no discernible problems; no explicit cache management of the $\mathbf{x}$ vector was required.

At the end of a typical solution process, the communication thread on each MPI process has performed O(1k-10k) communication iterations and each computation thread has performed a varying number of computation iterations (often O(100-1k) out of sync). All of this occurs in the time it takes to run just a handful of iterations of a Krylov Subspace solver such as GMRES.

One of the conditions for convergence of chaotic methods is that s (the difference in iteration-number between relaxations), in equation 6.2, is bounded. There is no explicit checking of the value of s, because a residual check is performed every 100 communication iterations, by the communications thread – providing a sufficient condition for convergence. Once convergence criteria are reached, the communication thread sets a flag signalling all threads to cease. There is no guarantee on the number of iterations that have been performed by each thread.

All of this implementation, including splitting A and B portions of matrices/vectors, organizing MPI communication patterns and the solution process itself, can be found in *Chaos*. Indeed, the C++ classes for vectors and matrices do not include any chaotic-specific code, and could be used to create a variety of hybrid-parallel solvers.

6.5 Verification

The matrix analysis tool, used to derive numerical properties from CFD matrices in section 6.2, was developed further to allow verification of linear solvers – the chaotic solver in particular. This verification uses matrices exported from ReFRESCO and matrices from the Matrix Marketplace [21].

The solution produced by the chaotic solver was compared against the solution produced by GMRES (with a restart of 3, 30 and 100), P-GMRES and BCGS. Preconditioners were not used and there was no attempt to compare performance, these tests were performed purely to compare the absolute solution and verify the implementation. The solvers were required to iterate until a desired absolute residual in the ℓ^2-norm.

Three analytical matrices from the matrix marketplace were tested ($pde225$, $pde900$ and $orsirr\_1$), along with a pressure equation matrix extracted from a 14.4k-cell lid-driven cavity flow simulation ($LDCF$). The full results for the smallest case ($pde225$) are shown in figure 6.3, and partial results are shown for the other matrices in figure 6.4. The results were similar for a range of convergence tolerances from 10^{-1} to 10^{-7} (not shown).

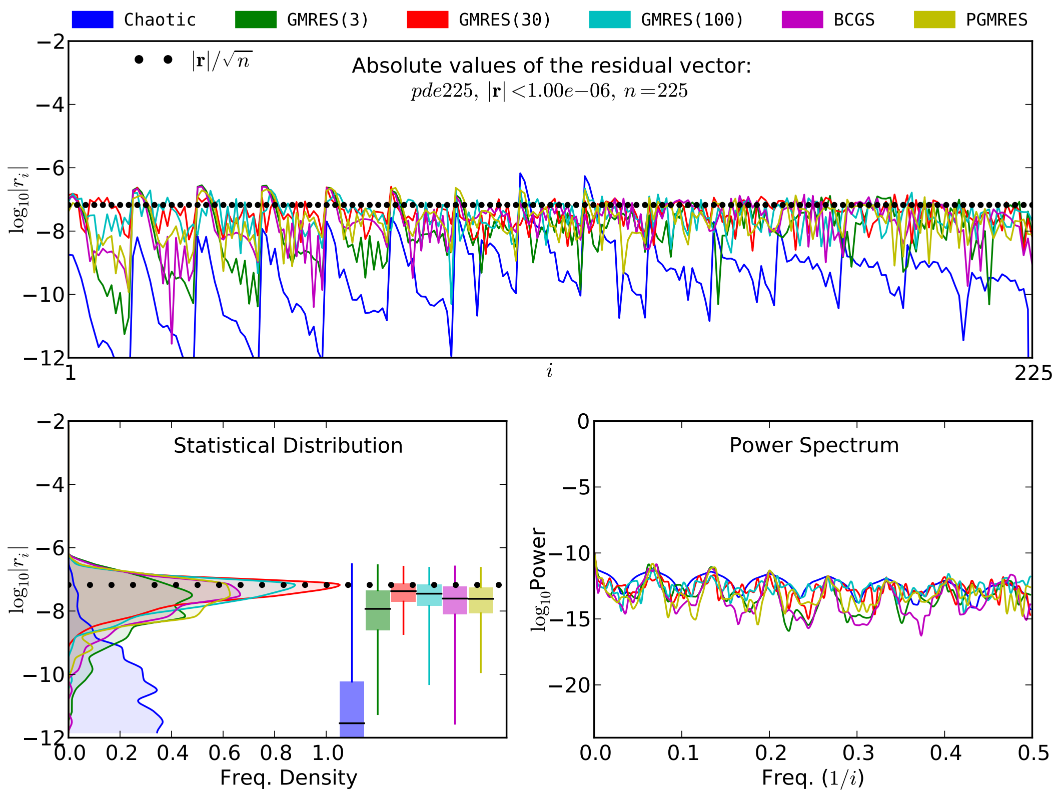


Figure 6.3: Full verification results from the *pde225* case, with a requested absolute ℓ^2-norm of 10^{-6}. [Top] A plot of the raw residual vector ($\mathbf{r} = \mathbf{Ax} - \mathbf{b}$). [Bottom-Left] Statistical Distribution of the residual vector, showing the mean, upper/lower quartiles, and minimum/maximum values. [Bottom-Right] Spectral analysis of the residual vector, with peaks corresponding to frequencies of error in the converged system. The dotted lines in the results mark the mean value of the residual required in order to satisfy the ℓ^2-norm. This line divides the elements of the residual that positively or negatively contribute to the ℓ^2-norm.

The residual vector $\mathbf{r} \in \mathbb{R}^N$ itself was plotted, in order to observe any anomalies in the solution. Following this, a distribution of the residual values was plotted, which revealed some differences in the solvers' behaviour; and finally, spectral analysis was performed on the residual vector to note the frequency of errors, to ensure that that no particular frequencies were left under-solved.

Although there are some differences in the solution vector, there are no concerns for any of the solvers. There is a tendency for the chaotic solver to over-converge the system (particularly on the smallest case, $pde225$), presumably because many relaxations can occur between residual updates. The KSP methods compute residuals each iteration, so this does not happen.

6.6 Scalability Experiments

The results from chapter 5 are repeated, using the LDCF and KVLCC2 test case of approximately $2.68m$ cells – this time comparing preconditioned FGMRES, the

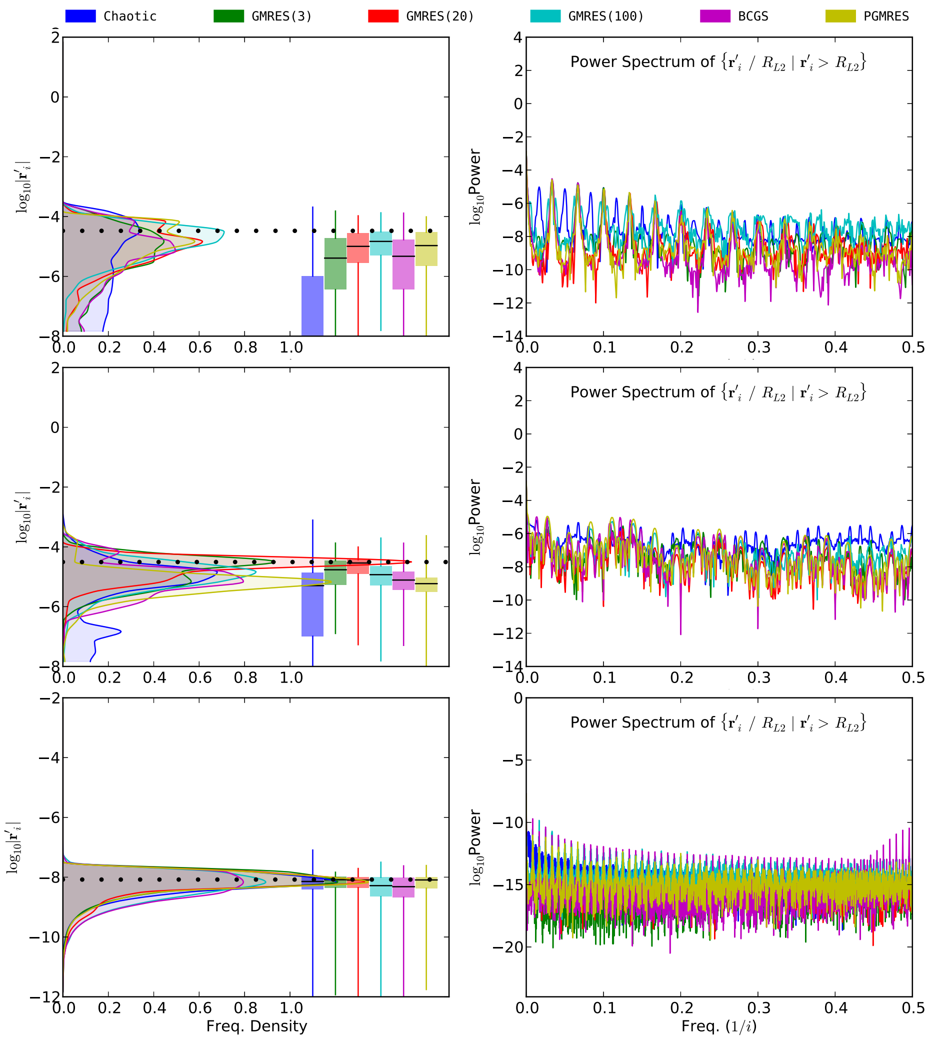


Figure 6.4: Partial verification results from the analytical cases *pde900* and *orsirr_1*, and the CFD case (LDCF) respectively, with a requested absolute ℓ^2-norm of 10^{-3} or 10^{-6} for the LDCF-case.

aforementioned SOR solver, and the chaotic solver. Scalability tests were performed up to 2048 cores. 100 outer-loops were performed, and the time spent in the solve routines for the pressure equation (per outer-loop) was measured. In these experiments, the scalability is normalized to T_8 (8-core performance), as this corresponds to a single MPI process of the chaotic solver. It is worth noting that the chaotic solver is identical, in all respects, to Gauss-Seidel, if it were run on a single core $C = 1$. Figure 6.5 shows the results of this scalability study, and table 6.2 gives a numerical representation of the same.

As before, the LDCF test case appears much easier to solve than the KVLCC2 test case, for all solvers and convergence tolerances. The chaotic solver is faster than the plain SOR solver in all cases, with the gap widening as the equations become easier to solve – reaching a speed-up of $\times 37.8$ on LDCF at 0.1 ILCT, and $\times 11.0$ on KVLCC2 at 0.01 ILCT ($C = 2048$). Scalability of the chaotic solver is better than the SOR method,

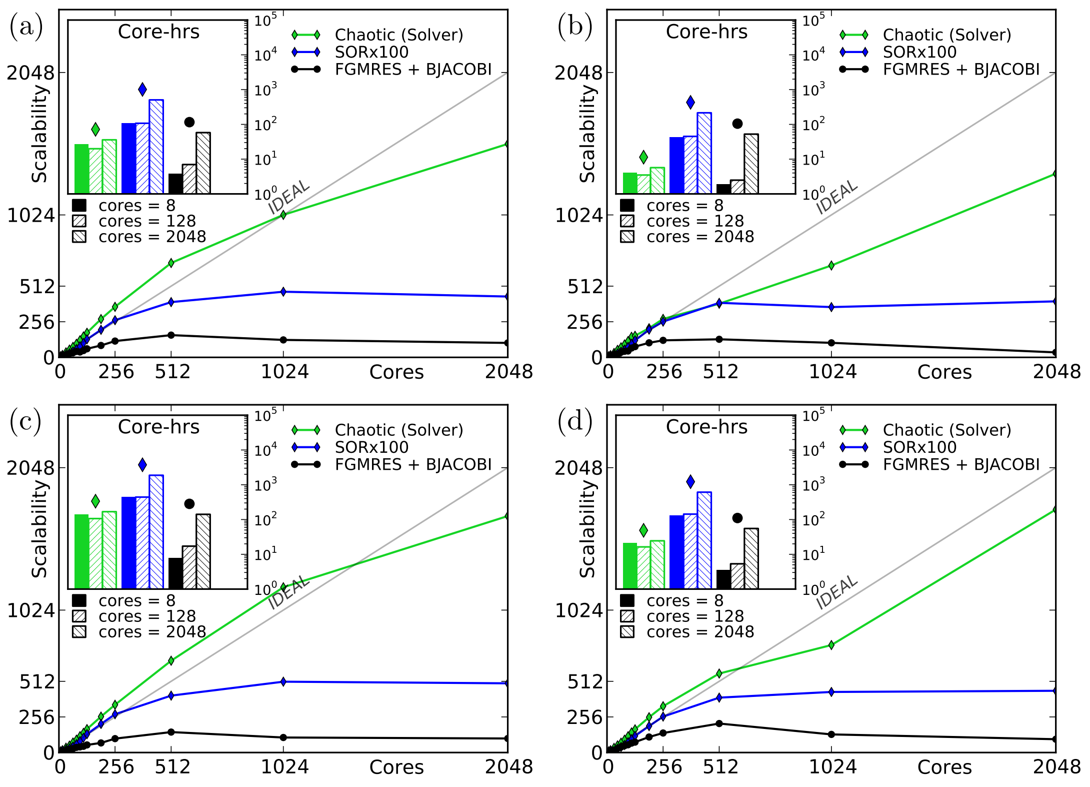


Figure 6.5: Scalability of the linear equation-system solver for the KVLCC2 (left) and LDCF (right) test cases, at 0.1 (top) and 0.01 (bottom) convergence tolerance.

Table 6.2: Absolute core-hours spent solving the linearized pressure equation over 100 non-linear iterations.

Case (ILCT)	FGMRES			SOR			Chaotic (solver)		
	$C = 8$	128	2048	8	128	2048	8	128	2048
LDCF (0.1)	1.8	2.5	54.4	39.5	43.7	211.5	3.6	3.4	5.6
LDCF (0.01)	3.4	5.3	56.8	127.9	146.2	618.4	20.7	16.9	24.5
KVLCC2 (0.1)	3.8	7.1	60.1	104.5	108.1	489.0	27.2	20.8	36.6
KVLCC2 (0.01)	7.8	17.4	144.0	442.2	447.3	1870.0	139.9	111.6	169.6

and is sometimes super-linear due to increasing use of the cache as the cells-per-core ratio decreases. Above 512 cores, particularly on the easier equation-systems, the scalability of the chaotic solver begins to deteriorate slightly. Obtaining computing time to thoroughly debug this was not possible, but it is suspected that the residual-check interval needs to be increased at this point. This is partly due to the decreasing cells-per-core ratio; and partly due to the increasing cost of global communications. Although the computation threads continue to iterate whilst waiting for the residual check, the communication thread (and vector-scatter updates) are stalled, reducing convergence rates.

Despite this, on all but the hardest equation-system, the chaotic solver is able to out-perform preconditioned FGMRES on 2048 cores, due to the poor scaling of the Krylov Subspace method. Indeed, on LDCF at 0.1 ILCT, the chaotic solver is almost ×10 faster than FGMRES. On stiffer equation systems, the gains are smaller because more low-frequency errors must be reduced (FGMRES is ×1.17 faster than chaotic on KVLCC2 at 0.01 ILCT). SOR and the chaotic solvers are both ill-equipped to deal with these low-frequency errors, especially as N increases to more practical levels. This is particualy noticable on a lower number of cores, where the scalability of FGMRES is still good.

6.7 SIMD Vectorization Optimization

Since the core of the chaotic algorithm is just sparse-matrix-vector-multiplication (SpMV), it is logical to optimize the data layout for this kernel. Experiments were performed with row-reordering and padding, to create highly-vectorizable loops which could take advantage of SIMD (single-instruction-multiple-data) optimization. Vectorization is already performed by the compiler, whereby $a_{ij} \times x_j$ element multiplications in each relaxation can be vectorized. However, it is inefficient because the matrix rows are often short and of variable length.

Padding the matrix rows with zeroes, to force equal lengths, did not provide any noticeable improvement, despite reducing the branching of vectorized loops. At worst, the additional unnecessary work of multiplying zeroes reduced performance. An optimization of this method was performed, where rows were reordered into groups with the same non-zero pattern (i.e. all rows with 4 non-zeroes per row are looped separately). This format is known as Jagged Diagonal (JAD); the overhead made this not worthwhile.

JAD is usually employed so that SpMV can be performed in a column-major way. The vectorizable dimension in column-major SpMV is of length n – much larger than the row-major format (where the maximum dimension is the number of non-zeroes per row) – granting better computational performance. However, since a relaxation is not complete until every column has been visited, the relaxations must occur in a Jacobi-like format (updates cannot be used as they are computed as in Gauss-Seidel). This spoilt the numerical performance of the chaotic solver, requiring many more iterations to achieve the same convergence. Staggering of these row-major relaxations with striding or blocking techniques were tried, but with no significant improvement.

Overall, the original implementation using the classic row-major compressed row storage (CRS) method could not be beaten significantly and consistently. The vectorization speed-up remains at approximately 2.2 out of a theoretical 4.0 on Iridis4. It is assumed that this is limited by the indirect memory access of x_j rather than the data structure of the matrix – this is a property of unstructured CFD codes which cannot be circumvented.

6.8 Scheduled Relaxation

The biggest problem with the chaotic solver, and other Jacobi-like solvers, is their inability to converge the lowest-frequency errors in the flow faster than $O(N^3)$ [103]. This can be overlooked when N is small, and if ILCT is high then convergence can often be reached by only smoothing high-frequency errors. To be competitive at solving the pressure equation in practical CFD simulations, the numerics of the chaotic solver must be improved. A recent study by Yang & Mittal [106], entitled *Acceleration of the Jacobi Iterative Method by Factors Exceeding 100 Using Scheduled Relaxation*, provides an interesting start point.

Typically, an over-relaxation parameter ω cannot be applied to iteration matrices whose spectral radius, ρ, is close to unity. If $\rho\omega > 1$, then the solution will diverge. The wave-number associated with with the largest eigenvalue would be over-relaxed so as to cause instability and divergence. In Yang & Mittal [106] the author shows how a relaxation schedule can be created to achieve the best of both worlds. If a relaxation parameter ω_1 is chosen at one iteration, it will rapidly converge a particular wave number in the system whilst attenuating error in other wave numbers. An under-relaxation parameter ω_2 is then used, for several iterations, to reduce the error in the unstable wave numbers. Multiple layers of this method can be used – for example, 4 different over-relaxations may be chosen, scattered in between many under-relaxations. The author describes how to create a schedule which is guaranteed to converge for the Jacobi method. The over-relaxation values may be very large, often $\omega > 100$.

A simple version of Scheduled Relaxation Jacobi (SRJ) was implemented and briefly tested as part of this PhD. On simple test cases the SRJ method performed up to 20-times faster than plain Jacobi. Furthermore, an attempt was made to couple the chaotic solver with the SRJ method, creating an 'SRC' scheme, but with some nuances.

The chaotic solver behaves similarly to Gauss-Seidel, in that new values of $\mathbf{x^k}$ are used the moment they are computed (in the same, k^{th} iteration). In Yang & Mittal the schedule is carefully chosen to avoid over-relaxing consecutive iterations, otherwise the error associated with certain frequencies would overflow the floating-point limits of the computer. In Gauss-Seidel, over-relaxing x_1^k and immediately using this value to over-relax x_2^k, and so on, causes a floating-point overflow almost immediately. For over-relaxation, a Jacobi-type relaxation must be used, so that each relaxation is using non-over-relaxed values of x_j^{k-1}.

In the SRC scheme, the over-relaxations (of which there are few) are performed using a standard synchronized Jacobi iterations. The many intermediate under-relaxations are performed using a truly chaotic solver. It is difficult to guarantee that enough under-relaxation has been performed, since it is not possible to simply count iterations, but it is possible. The details of this scheme, for guaranteeing enough under-relaxation, will be explored further in the following chapter – where it is required for multigrid methods.

A modest 4.1-times speed-up was achieved with SRC, compared to the pure chaotic solver, on a simple test cases (3D LDCF, 1-million cells, 16 cores). However, there are some fundamental problems. Construction of the relaxation schedule requires assumed information about the maximum and minimum eigenvalues occurring in the system. For uniform grids (such as LDCF) this is not such a problem, but for realistic CFD simulations scheduled relaxation becomes impractical. Furthermore, the maximum acceleration is limited by the largest cells in the domain; for flows with large far-field elements and very small boundary elements, the speed-up may be limited.

In the literature, no similar work has been done on asynchronous or chaotic methods with scheduled relaxation. If scheduling can be generalized to unstructured grids successfully, then the coupling of scheduling and chaotic methods could be revisited.

6.9 Summary & Conclusions

Overall, the chaotic solver shows excellent results compared to SOR, reaching speed-ups between $\times 11.0$ and $\times 37.8$; and is even capable of out-performing KSP methods on a large number of cores (almost $\times 10$ faster, due to poor scaling of FGMRES). As N increases, the rate of convergence for SOR and chaotic solvers deteriorates with $O(N^3)$, which means these solvers are unsustainable for practical computations. Attempts to circumvent this $O(N^3)$ limitation using scheduled relaxation were unfruitful.

As was noted in chapter 5, multigrid methods have strong numerical properties but often struggle with scalability. Much of this poor scalability can be attributed to the smoother, so there may be potential to replace classical SOR smoothers with a chaotic smoother, hopefully transferring the speed-ups proven in this chapter. The following chapter explores this possibility.

Chapter 7

Chaotic-Cycle Algebraic Multigrid

In this chapter, the intention is to apply the principle of chaotic relaxations to a multigrid method, gaining the scalability benefits of chaotic methods – particularly on the coarsest grids of the multigrid algorithm where the cells-per-core ratio is very low. A generic multigrid algorithm has been developed below, employing an unsmoothed-aggregation technique and the classical V-, W- and F-cycles. Following this, the extension to a 'chaotic-cycle' multigrid is explained, where the boundedness of s and the removal of other synchronization points is explored. Verification of $O(N)$ performance is shown on a generic Poisson equation. Finally, scalability and performance of all the multigrid cycles are tested against the standard chaotic solver, FGMRES and ML. This work closely follows the latter half of Hawkes et al. [40], available as appendix E.

7.1 The Theory of Algebraic Multigrid

The principle of algebraic multigrid methods is that low-frequency errors can be solved quickly by translating the problem onto a coarser grid, where low-frequency errors become high-frequency errors. This coarsening can be applied recursively over many levels. Smoothers such as SOR can be used to quickly smooth high-frequency errors on the coarse grids. Often, only a single iteration of the smoother is required. It can be shown that the number of multigrid iterative cycles required to converge a perfect multigrid method remains constant as N increases, which makes them competitive against Krylov Subspace methods and relevant for real-world applications. In theory, the work done in each multigrid iteration scales with N, since all of the relevant routines (such as smoothing) are $O(N)$. Thus, a multigrid solution should take $O(N)$-time, although this does not include the time taken to construct the coarse grids, and also depends on the quality of the coarse-grid operators [103, §2.4][68].

There are certain traits of the chaotic solver which make it difficult to substitute as a smoother in a multigrid method. Most of these issues arise because it is difficult to impose stopping criterion and guarantee the boundedness of the asynchronicity s.

In the chaotic solver, a sufficient condition for convergence was provided by checking the residual – which was necessary anyway. For a multigrid method it is not necessary to compute a residual on the coarse grids, and it would be inefficient to do so, so an alternative method for bounding s is required.

Another issue is that smoothers only need to run for a small number of iterations, which does not suit chaotic methods well, especially when the equation-systems are small and easy to solve (as on the coarse grids). Chaotic solvers have a tendency to over-converge these systems considerably, because the computation threads achieve so much smoothing before a single communication can even occur. Although they can never be slower than existing smoothers because of this, much of this free convergence is wasted if it not required. Since there are other sources of synchronization in the multigrid cycle, it would be beneficial to trade off some of these synchronizations for additional smoothing, which can be obtained for free. Indeed, the idea of running a chaotic solver for one "iteration" is virtually meaningless!

Below, the implementation of a classical algebraic multigrid is considered. Following this, the extension to a chaotic-cycle multigrid is considered; and the above points – regarding bounded asynchronicity and removal of unnecessary synchronization – are tackled.

7.2 Implementation of Classical Multigrid

There are two stages to a multigrid algorithm. The first stage is concerned with constructing coarse grids and interpolation operators. The second stage is the multigrid 'cycle' which is the iterative process used to solve the equation system.

7.2.1 Construction of Coarse Grid Operators

In the coarsening stage, coarse grid operators are recursively created from a fine grid, such that:

$$\mathbf{A}_m = \mathbf{R}_m \mathbf{A}_{m-1} \mathbf{P}_m, \quad 1 \leq m \leq m_{max}$$

where $\mathbf{A}_m$ is matrix $\mathbf{A}$ on the m^{th} level (level zero being the finest, original matrix); $\mathbf{P}_m$ is the prolongation matrix, a sparse rectangular matrix which describes the interpolation between $\mathbf{A}_m$ and $\mathbf{A}_{m-1}$; and $\mathbf{R}_m$ is the restriction matrix, a sparse rectangular matrix describing the contraction from $\mathbf{A}_{m-1}$ to $\mathbf{A}_m$. $\mathbf{R}_m$ is usually the transpose of $\mathbf{P}_m$.

Here, an aggregation-type coarsening is used, loosely based on Notay [70]. In aggregation schemes, a coarse grid is constructed by grouping fine-grid elements into single coarse-grid elements (or 'aggregates'), such that the aggregates form a disjoint subset of the fine-grid elements. In unsmoothed aggregation, $\mathbf{R}_m$ and $\mathbf{P}_m$ are simply boolean matrices which never need to be explicitly stored. In smoothed aggregation, the prolongation and restriction matrices are smoothed such that coarse-grid elements may contribute to more than one fine-grid element. The simpler unsmoothed-aggregation is used here.

When constructing the coarse grids, it is important that aggregates are created in the direction of the 'smoothest error'. In other words, they must be constructed such that the maximum eigenvalue is successfully reduced on the coarser grid [68].

Notay [70] proposes a pairwise-aggregation scheme which attempts this. The algorithm couples pairs of elements which are 'strongly-coupled', which means they have large negative values in the off-diagonals that connect them. The algorithm attempts to construct pairs with maximum coupling, but minimum un-paired elements. The process is computationally expensive, constantly searching for the least strongly-coupled elements so that they can be paired first. This constant sorting of elements can be expensive, and a pairwise scheme can only coarsen by a factor of two – meaning that repeated application is necessary to achieve 4- or 8-times coarsening.

A cheaper, but less rigorous approach has been adapted from this scheme. A maximum aggregation size a_{max} is specified (for example, 8). For each element e_i in the fine grid (iterated in order), the off-diagonal values of the matrix $(\mathbf{A}_{m-1})_{ij,j\neq i}$ create a set of coupled elements e_j that can form aggregates. For each coupled element, the strength of the coupling is recorded as $v_j = -(\mathbf{A}_{m-1})_{ij}$. If the coupled element already belongs to an aggregate, the number of elements already in that aggregate is recorded as a_j, else $a_j = 1$. An aggregate is formed by selecting e_j with a minimal a_j, and then a maximal v_j. The aggregation of the two elements is abandoned if both elements already belong to aggregates, or if either element belongs to an aggregate which already has a_{max} elements. The algorithm achieves close to a_{max} coarsening in a single step, very cheaply.

At the end of the aggregation process, a set of aggregates $G_i, i = 1, ..., n_m$ exist, each with approximately a_{max} indices which map to $\mathbf{A}_{m-1}$. This mapping is equivalent to $\mathbf{R}_m$ and the tranpose mapping is created for $\mathbf{P}_m$. From this point, $\mathbf{A}_m$ can be computed as:

$$(\mathbf{A}_m)_{ij} = \sum_{k\in G_i}\sum_{l\in G_j}(\mathbf{A}_{m-1})_{kl} \quad (1 \leq i, j, \leq n_m)$$

The process is repeated recursively, with $m = m + 1$, until the coarsest grid reaches a specified size. Figure 7.1 shows an example of this coarsening on a two-dimensional lid-driven cavity flow domain. Investigating the quality of the aggregation is beyond the scope of this thesis, as it is an active area of research which is not specific to chaotic methods. The presented method appears to be robust enough for the purposes of these experiments.

The aggregation is done per MPI domain, such that each process has its own set of hierarchical grids (as in [70]). Aggregation is not performed across process boundaries, thus $\mathbf{A}_m$ in the above discussion is replaced by $(\mathbf{A}_A)_m$ for each MPI domain. $\mathbf{A}_B$ is never considered when aggregating. Also, when constructing the coarse grids, $(\mathbf{A}_B)_m$ is not explicitly constructed because it would require re-mapping the buffered storage of the parallel vectors, $(\mathbf{x_B})_m$. Vector-scattering on coarse grids occurs by mapping into the buffer of the finest grid, resulting in excess, inefficient MPI communication. A more rigorous aggregation scheme which performs coarsening across MPI domains may be

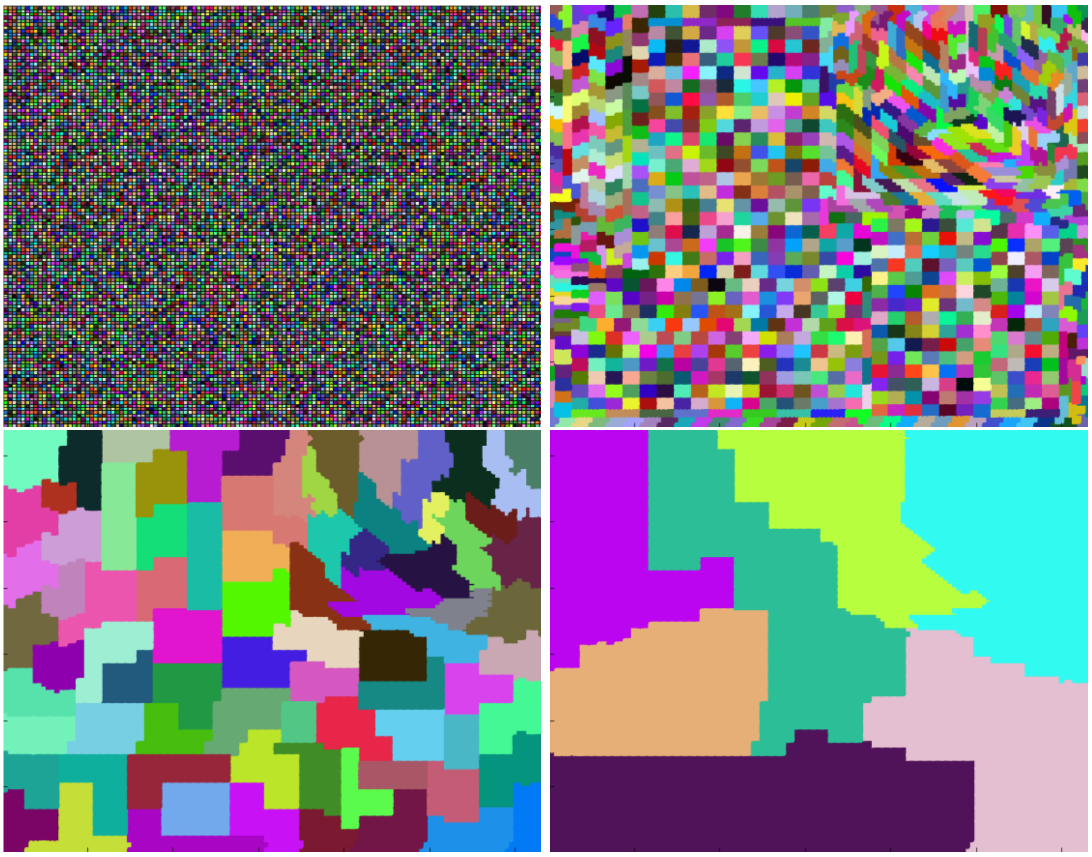


Figure 7.1: A view of the aggregation process, showing four multigrid levels with $a_{max} = 8$ on a two-dimensional lid-driven cavity flow domain. The fine matrix begins with $128 \times 128 = 16384$ degrees of freedom; and the coarsest matrix has just 7.

more performant; especially since this aggregation can be reused over several non-linear iterations of the overall CFD simulation.

7.2.2 Cycling Techniques

The second stage of the multigrid algorithm is the linear solution process, in which various different cycles can be used to visit the coarse grids. On each coarse grid, a 'correction' to the fine-grid solution is computed using the additive correction scheme [48]. There are three simple routines which allow a variety of cycles to be created.

- On each level, smoothing is applied to the linear system $\mathbf{A}_m\mathbf{x}_m = \mathbf{b}_m$. On the coarsest level a direct solver is often used, but a smoother will suffice too.
- Restriction is used to move from a fine grid $(m-1)$ to a coarse grid (m): from the linear system $\mathbf{A}_{m-1}\mathbf{x}_{m-1} = \mathbf{b}_{m-1}$, a new system is created $\mathbf{A}_m\mathbf{x}_m = \mathbf{b}_m$. $\mathbf{A}_m$ has been generated by the aggregation process, $\mathbf{x}_m$ is set to zero, and $\mathbf{b}_m$ is set to to the residual vector multiplied by the restriction matrix: $\mathbf{R}_m(\mathbf{b}_{m-1} - \mathbf{A}_{m-1}\mathbf{x}_{m-1})$. Smoothing performed before a restriction is called pre-smoothing.
- In the other direction, prolongation is used to apply the coarse-grid correction to the fine grid. The prolongation matrix, $\mathbf{P}_{m-1}$ is used to expand $\mathbf{x}_m$ into $\mathbf{x}_{m-1}$. The vector $\mathbf{P}_m\mathbf{x}_m$ is summed with $\mathbf{x}_{m-1}$, to provide a *correction* to the

fine grid. Post-smoothing is usually performed after prolongation to smooth the errors introduced by the unsmooth coarse-grid correction – smoothed-aggregation techniques reduce the amount of post-smoothing required.

These three routines, combined with different values of a_{max}, different minimum grid sizes, and different numbers of smoothing steps, can produce a wide variety of multigrid cycles, such as the V-cycle, F-cycle, W-cycle and sawtooth-cycle (figure 7.2).

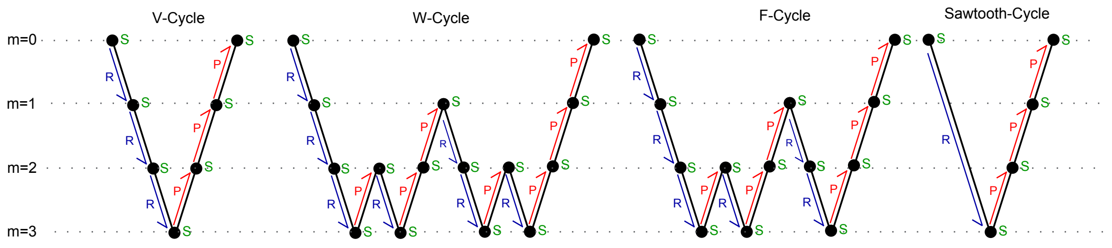


Figure 7.2: A view of the solution process, showing a single iteration of a four-level multigrid method using the classical recursive V-cycle, W-cycle, F-Cycle and Sawtooth-Cycle. Restriction, Prolongation and Smoothing are colour-coded.

7.3 Applying Chaotic Principles to Multigrid Methods

Consider the classical V-cycle as illustrated, using an SOR smoother. In each pre-smoothing step there is an implicit global synchronization during each iteration, as has been discussed in earlier chapters. For restriction, a vector-scatter is also necessary to compute $\mathbf{b}_m = \mathbf{R}_m(\mathbf{b}_{m-1} - \mathbf{A}_{m-1}\mathbf{x}_{m-1})$, causing another implicit global synchronization.

Replacement of the smoother with a chaotic smoother solves the implicit synchronization in the pre-smoothing steps, but since the smoother only needs to run for a handful of iterations, the synchronization required for restriction still has a large effect. In order to minimize this, consider a sawtooth cycle, which is simply a V-cycle with no pre-smoothing – the right-hand sides ($\mathbf{b}_m$) of the all the coarse matrices must still be initialized. In the 4-level example shown in figure 7.2, communication of $\mathbf{x}_0$ is required as usual to create $\mathbf{b}_1$, requiring a vector-scatter. $\mathbf{x}_1$ is reset to zero. The next restriction follows immediately to create $\mathbf{b}_2$; except now it is known that $\mathbf{x}_1 = 0$ there is no need to scatter $\mathbf{x}_1$, and $\mathbf{b}_2 = \mathbf{R}_2(\mathbf{b}_1 - \mathbf{A}_1\mathbf{x}_1) = \mathbf{R}_2\mathbf{b}_1$. It follows that the entire sawtooth restriction requires only one implicit synchronization. The downside of using a sawtooth method is that more post-smoothing iterations may be required. For the chaotic smoothers this may not be a disadvantage, because much of this smoothing can be obtained for free.

Returning from the coarse grid to the fine grid, post-smoothing with a chaotic smoother does not require any implicit synchronization. Furthermore, prolongation does not require any synchronization either; because $\mathbf{x}_{m-1} \mathrel{+}= \mathbf{P}_m\mathbf{x}_m$ does not require off-process values of $\mathbf{x}_m$ (it is an entirely local operation). Smoothing on level $m-1$ can begin before other processes have performed their own prolongation, because the off-process values of $\mathbf{x}_m$ that are not updated are still zero. Updated values will be provided by communications in the chaotic smoother when they are ready, but chaotic relaxations can begin without that information.

That being said, it is still necessary to guarantee the boundedness of s in the chaotic-cycle prolongation phase. Consider the situation in which p post-smoothing iterations are requested at each level, and this is expected to provide enough convergence on the coarse grids to guarantee overall convergence of the multigrid method. Note that a certain amount of post-smoothing must be performed in order to compensate for unsmooth prolongation [103], else the multigrid solver will not converge. For the chaotic smoother, specifying a number of iterations is meaningless because one process could complete its iterations before receiving updates from any other process or thread, and parts of the equation-system would not be smoothed correctly. Thus, a method for bounding s has been developed:

The chaotic-cycle, like the chaotic solver, is designed for a hybrid MPI+OpenMP environment. The same concept of computation and communication threads is used, and the communication threads remain in sync across MPI processes. Within each MPI process, each thread stores an iteration counter k in an array, initialized to $k = 0$ at the start of each smoothing phase. After each computation thread completes a chaotic relaxation on its portion of the equation system, it attempts to increment k – however, it is only allowed to increment k if the values in the counter array are all $\geq k$. If any value is less than k, it suggests that the relaxations may have used old values and therefore it did not 'count' towards p. This bounds s between computation threads. The communication thread also participates in the same counter array, attempting to increment k every time it scatters $\mathbf{x}_m$ between MPI processes; thus bounding s between all processes. To avoid an MPI deadlock, additional checks are required to make sure communication threads perform the same number of iterations on all processes (regardless of k); this is done using non-blocking MPI barriers (`MPI_Ibarrier`). This method was developed earlier in the PhD for use in chaotic scheduled relaxation methods (discussed in section 6.8), in order to guarantee a certain number of meaningful under-relaxation iterations.

It is unlikely that any computation thread will complete exactly p iterations, since they are usually faster than communications, but it is the absolute lower bound. Most threads will complete many more than p iterations, so choosing a value for p is not as intuitive when compared to classical multigrid cycles.

Combining this bounded chaotic smoother with the sawtooth cycle and barrier-less prolongation: a chaotic-cycle multigrid is created. The entire cycle has only one implicit global synchronization, required to restrict the finest level. Instead of explicitly scattering $\mathbf{x}_0$ the latest values communicated during the pre-smoothing step are used, further hiding this implicit synchronization behind useful relaxations. One explicit thread-barrier (local) is also required at the end of the restriction simply to signal that all of the coarse grids have been initialized, because it is important that $\mathbf{x}_m$ has been reset to zero on every level before prolongation begins – but this is cheap. During each multigrid cycle, a lagged, non-blocking residual is computed to determine an overall stopping criteria.

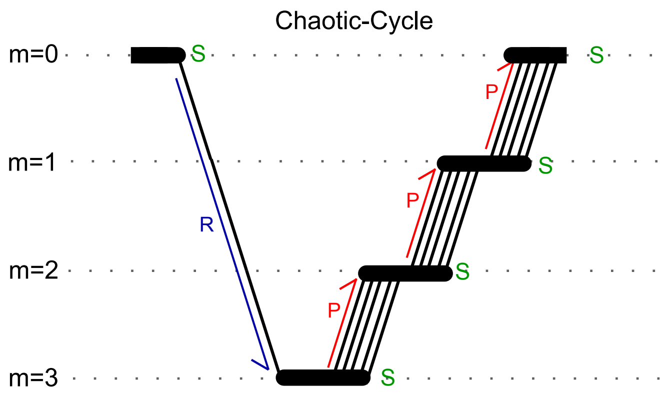


Figure 7.3: A view of the chaotic-cycle solution process, showing a single iteration of a four-level multigrid method. Restriction, Prolongation and Smoothing are colour-coded. Multiple parallel lines are shown during the prolongation stage to depict the non-deterministic way in which multiple threads (including communication and computation threads, across multiple processes) may participate in the chaotic-cycle.

Figure 7.3 shows an illustration of the chaotic-cycle, although it is difficult to capture the chaotic nature in a simple diagram. There are certainly other ways in which chaotic methods could be applied to create a scalable multigrid cycle – possibly involving smoothing on multiple levels simultaneously – but this algorithm is the most obvious to pursue and does not break the standard convergence theorems for multigrid solvers.

7.4 Verification

In order to ascertain that the chaotic-cycle conforms to the convergence expectations of a multigrid method, verification is performed using a generic 3-dimensional (i, j, k) Poisson equation with Dirichlet boundaries. A 3D, unit-length, cubic domain of N elements is created (element spacing $h = 1/N^{\frac{1}{3}}$). A pre-determined solution $\mathbf{x}_s$ is created for the linear system, by combining 20 sinusoidal frequencies: $\sum_{f=0}^{19} \sin(2^f \pi.i.j.k.h^3)$. $\mathbf{b}$ is created from this solution and $\mathbf{x}$ is initialized to zero. The source is included as an example in *Chaos*.

The number of iterations to convergence (relative tolerance 10^{-6}) and time-per-iteration is measured for the chaotic-cycle and V-cycle as N increases from 512 (8^3) to 1.73-million (120^3). With perfect coarsening, the number of iterations to convergence should be constant, with the time-per-iteration increasing linearly with N. However, this ideal situation cannot be achieved with piecewise-constant unsmoothed aggregation [68]. Figure 7.4 shows the results of this verification. The experiment is performed on 64 cores (8 MPI processes with 8 threads each). The aggregation factor, a_{max}, is set to 8, and the number of pre- and post-smoothing iterations is set to $p = 3$.

The number of iterations required to reach convergence is not quite constant, but appears to approach an asymptotic maximumum, for both cycles. This suggests small imperfections in the aggregation process, which is expected. The time-per-iteration is $O(N)$ as desired, although both the V- and chaotic-cycle show occasional deviations or noise. This is probably related to the aggregation factor and the discontinuous number of levels being created as N increases.

Overall, the two cycles perform similarly; but these tests are not designed to gauge performance. The important result here is that the chaotic-cycle is at least as good as the V-cycle from the perspective of multigrid correctness. The following section properly assesses the performance and scalability of the various cycling techniques.

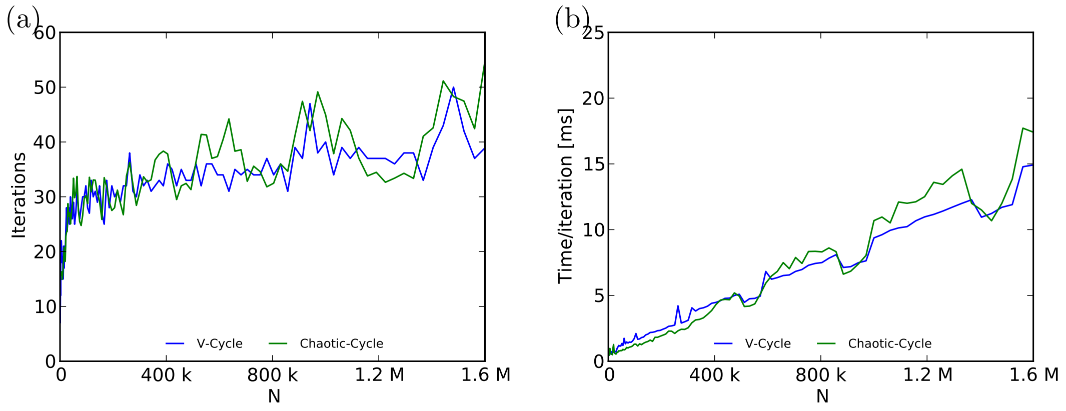


Figure 7.4: Number of iterations (a) and time-per-iteration (b) to converge the discrete Poisson equation, of size N, to a relative convergence tolerance of 10^{-6}.

7.5 Scalability Experiments

The performance experiments from the previous chapter are repeated here, this time comparing the chaotic-, V-, W- and F-cycle with preconditioned FGMRES, the standard chaotic solver, and a state-of-the-art multigrid package (ML, [29, 41]). Once again the solution of the pressure equation is summed over 100 outer-loops of the CFD solver. For all multigrid methods, including ML, aggregation is only performed once in this time, and then re-used. This is possible because the non-zero structure of the matrix $\mathbf{A}$ does not change between outer loops, thus the restriction and prolongation matrices do not need recomputing. $\mathbf{A}_m$ must still be refilled on the coarser grids using equation 7.2.1. Aggregation and re-filling time is included in the results. Occasional re-aggregation is probably required for practical simulations, because the coupling of various elements in the fine grid may change as the overall CFD solution converges, and the coarser grids will no longer follow the smoothest errors.

Preliminary studies showed that p (the number of pre- and post-smoothing iterations) had little effect on the results within a range of 1 to 12. Increasing p resulted in fewer cycles being required overall (and vice versa), but with almost no effect on scalability and absolute speed. $p = 3$ was selected for all of the *Chaos* multigrid methods; and $p = 1$ was used for ML. ML uses a complex smoothed-aggregation coarsening strategy which achieves smoother coarse-grid-corrections, thus less smoothing is required. An additive-correction V-cycle was used in ML, with a direct solver on the coarsest grid.

The results are shown in figure 7.5 and (partially) table 7.1. Overall, the scalability of the V-, W- and F-cycle multigrid is poor, and somewhat similar to the Krylov Subpsace method (FGMRES). The V-, F- and W-cycle all perform similarly on $C = 2048$ cores.

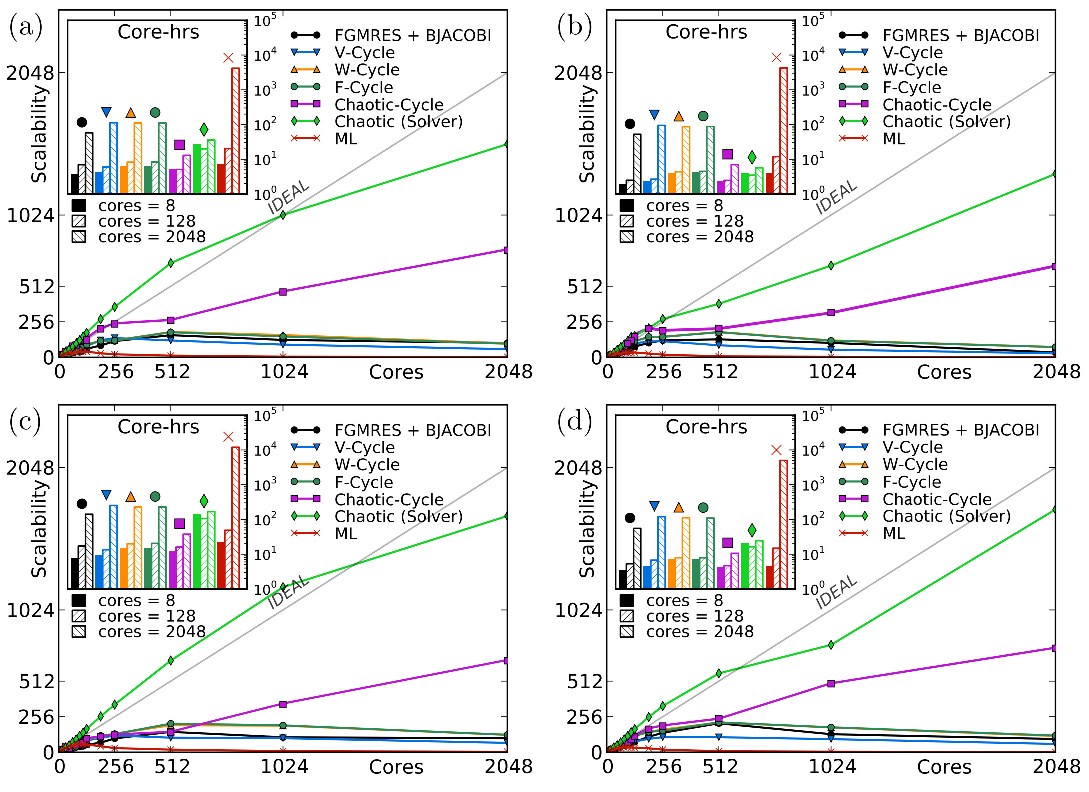


Figure 7.5: Scalability of the linear equation-system solver for the KVLCC2 (left) and LDCF (right) test cases, at 0.1 (top) and 0.01 (bottom) convergence tolerance.

Table 7.1: Absolute core-hours spent solving the linearized pressure equation over 100 non-linear iterations.

Case (ILCT)	FGMRES			V-Cycle			Chaotic-Cycle			Chaotic (solver)		
	$C = 8$	128	2048	8	128	2048	8	128	2048	8	128	2048
LDCF (0.1)	1.8	2.5	54.4	2.3	2.7	94.6	2.3	2.5	7.1	3.6	3.4	5.6
LDCF (0.01)	3.4	5.3	56.8	4.3	6.8	124.1	4.0	4.7	10.6	20.7	16.9	24.5
KVLCC2 (0.1)	3.8	7.1	60.1	4.1	6.1	117.2	5.1	5.3	13.6	27.2	20.8	36.6
KVLCC2 (0.01)	7.8	17.4	144.0	9.2	13.9	258.9	12.4	16.3	38.5	139.9	111.6	169.6

On a smaller number of cores, the size of $\mathbf{A}_A$ in each domain becomes larger, thus the total number of levels increases and the algorithmic differences between the three cycles become larger. As the number of levels increases, the W- and F- cycles spend exponentially more time visiting coarse grids, and perform worse than the V-cycle. If only two grids existed, all three cycles would be identical. This effect makes it difficult to differentiate between numerical effects and scalability of the classical multigrid cycles.

The chaotic-cycle exhibits slightly worse absolute performance on $C = 8$ than the V-cycle and FGMRES, but offers much greater scalability. In all cases the chaotic-cycle is the fastest solver on $C = 128$ cores. On the most difficult test case (KVLCC2 at 0.01 convergence tolerance), the chaotic-cycle is over 6.7× faster than the classical V-cycle on $C = 2048$ cores, and 3.7× faster than preconditioned FGMRES. This gap widens on easier equations, reaching a 13.3× and 7.7× speed-up over V-cycle and FGMRES, respectively, on LDCF at 0.1 convergence tolerance.

The scalability of ML was poor in all cases. It was suspected that the rigorous aggregation process is a bottleneck to scalability when such a lax convergence tolerance is specified; however, re-use of the aggregation had little effect on scalability. Other efforts to experiment with ML were unsuccessful in improving scalability – including adjusting the number of multigrid levels, the amount of smoothing, and replacement of the direct solver with a smoother on the coarsest grid.

Despite its good performance, the chaotic-cycle multigrid falls short of ideal scalability. Indeed, in one case the simple chaotic solver is faster than chaotic-cycle multigrid due to its superior scaling.

7.6 Discussion

Both the classical- (V,W,F) and chaotic-cycle should achieve better scalability if the vector-scattering on coarse-grids is improved. Currently, the MPI buffers and indexing of $(\mathbf{x}_B)_0$ and $(\mathbf{A}_B)_0$ are also used on the coarse grids, instead of creating smaller, aggregated buffers for each level, which is inefficient. Computing buffers for each coarse-grid would certainly improve scalability. It is likely that the chaotic-cycle hides some of these losses (since extra relaxation can occur whilst waiting for these communications), so the differences between the two solvers may become smaller.

Perhaps a bigger concern is (once again) the residual computations which currently occur asynchronously during each multigrid iteration. On 2048 cores the aggregation process only creates four levels, and with $p = 3$ a chaotic-cycle will perform a minimum of 18 vector-scatters in this time. As noticed in the chaotic solver, with a residual computed every 100 iterations this was already the main limitation to overall scalability. It is suspected that this is the main cause of poor scalability and is also the only sensible reason that the standard chaotic solver could out-perform the chaotic-cycle multigrid (since the chaotic-solver is equivalent to a one-level chaotic-cycle multigrid method). For both the chaotic-cycle multigrid and chaotic solver, the ideal solution would be to apply a threaded residual computation check, such that residuals are only computed at the speed at which they can be communicated. In this way the residual computation could never stall the communication thread, but it would also never lag unnecessarily. In some ways this could be considered a 'chaotic' residual.

The problem with such a scheme is that it is difficult to stop MPI processes safely without some form of explicit, predictable barrier or global communication – because if any process issues `MPI_Isend/MPI_Irecv` commands before receiving the 'stop' notification, these unmatched communications will cause a deadlock. As mentioned earlier, some form of one-sided communication would be ideal to allow communication threads to operate chaotically. This would also allow chaotic residual computations to be implemented easily.

Throughout this thesis, it has been easy to peer into the internal workings of each solver using profiling tools such as Score-P. However, profiling the chaotic-cycle multigrid is somewhat difficult. The numerical and computational properties of both the chaotic solver and chaotic-cycle multigrid are so closely intertwined, it can be hard to distinguish between numerical and computational issues. Indeed, the whole purpose of chaotic methods is to make use of all available hardware for useful numerics, even when some systems (such as communications) are struggling. Furthermore, any attempt to intrusively debug or profile these solvers can affect the non-deterministic solution path of the solver. This inability to easily debug and profile is possibly the main disadvantage of chaotic methods – at least from a research and development perspective.

Even though it can be seen that both the coarse-grid communications and residual checks are slow, it is not clear which is the main source of performance losses. However, it is suspected that the residual issue is far greater, because whilst waiting for the residual to complete the communication threads are blocked from updating the communication buffers ($\mathbf{x}_B$) on the finest grid. At least on the coarse grids, $\mathbf{x}_B$ is periodically updated, providing fresh values for the excess relaxations to work on – even though the communications are slow.

It is likely that both of the aforementioned issues affect the V-, W- and F-cycle too. Indeed, since they do not have the benefit of performing chaotic relaxations they may be more affected. Nonetheless, the chaotic-cycle multigrid demonstrates good scaling compared to these classical cycles; and its ability to weather ineffective communications should allow it to perform well in many-core environments.

7.7 *Chaos* on Xeon Phi

As a brief aside, *Chaos* was compiled and tested on a single (non-MPI) Xeon Phi co-processor (5110P), exploiting up to 60 physical cores and 120 OpenMP threads. It was not possible to compile the entire ReFRESCO suite for the Xeon Phi – mostly due to external dependencies; instead, some performance tests were performed on the generic Poisson case introduced in section 7.4. The convergence tolerance was set to 10^{-6} and the time-to-convergence was measured for the V-cycle and chaotic-cycle, shown in table 7.2. There is no MPI partitioning; thus there is a single multigrid hierarchy (rather than one per MPI process) – this is the main numerical difference between the experiments performed in section 7.4.

Table 7.2: Performance of V-cycle and Chaotic-cycle multigrid from *Chaos* on a single 5110P Xeon Phi. The time required for aggregation is ommitted from these results.

N	V-Cycle	Chaotic Cycle
8k	25ms	26ms
64k	55ms	51ms
1.73m	2169ms	1512ms

As N increases the chaotic-cycle out-performs the V-cycle multigrid, suggesting strong performance on many-core architectures. According to Intel [50] a Xeon Phi 5110P can expect to outperform a single CPU node (2x E-2670) by a factor of 1.77 (for general fluid dynamics benchmarks) to 2.38 (Linpack benchmarks)[1]. Using a single CPU

[1]Conveniently, nodes in Iridis4 consist of 2x E-2670, although other hardware (i.e. regarding memory bandwidth) may be different.

node, the Chaotic-cycle takes $2452ms$ to converge the largest case shown in table 7.2, thus the Xeon Phi achieves a 1.62-times speed-up. This is a promising result, but much more research should be performed in this area.

7.8 Conclusion

This chapter has concerned itself with the development of a chaotic-cycle multigrid algorithm, which demonstrates the application of chaotic relaxation theory to a typical multigrid method.

The chaotic-cycle multigrid outperforms all other solvers (including Krylov Subspace solvers, classical multigrid methods, and state-of-the-art smoothed-aggregation multigrid methods) in terms of scalability and absolute speed, from 128 cores upwards. Below this, the chaotic-cycle multigrid is still highly competitive. On 2048 cores the chaotic-cycle multigrid is up to 7.7$\times$ faster than Flexible-GMRES and 13.3$\times$ faster than classical V-cycle multigrid. Scalability of the chaotic-cycle multigrid can still be improved. Indeed, on the easiest equation system the simple chaotic solver was faster (on 2048 cores) and this is indicative of an issue with too-frequent residual checks. $O(N)$ performance of the chaotic-cycle multigrid solver has also been verified.

The potential of chaotic methods compared to synchronous methods has been clearly demonstrated. In the following chapter, the techniques implemented in *Chaos* will be applied to a 'real-world' CFD simulation. The intention is to demonstrate that non-deterministic linear solvers can be used safely for CFD, without any negative affect on the overall flow results.

Chapter 8

Suitability of *Chaos* for CFD

In the previous chapter, the performance and scalability of a chaotic-cycle multigrid method was tested, showing promising results. However, there are still some remaining concerns which should be addressed in order to gain complete confidence that the chaotic-cycle multigrid can be used in everyday CFD.

These concerns are:

1. After thousands of applications of a chaotic, non-deterministic solver, is there any adverse effect on the overall fluid flow solution? Thus far, verification has been performed on a per-solve basis, where it has been shown that the actual solution vectors from various linear solvers can be quite different, despite achieving the same residual-norm. It is not clear whether this will be noticeable in the final flow solution.
2. Does the unsteadiness of the flow alter the performance of the linear solver? Until this point, all of the test-cases have been steady-state. The addition of unsteady terms is likely to make the equations less stiff, so has not been a major concern. However, a large difference between the two steady cases (LDCF and KVLCC2) has already been noticed, so it is worth explicitly testing an unsteady test case.
3. Is the absolute performance of the chaotic-cycle multigrid scaling correctly with $O(N)$? Will the performance translate to large, practical grids? This has already been tested on the pragmatic Poisson test case in the previous chapter, but a proper performance analysis on 'real' CFD equations will be useful.

In order to address these concerns, a highly-unsteady test case of a submarine at a $20°$ yaw angle will be used. The simulations are performed on three grids: $N = 2.78m$ (similar to earlier test cases), $N = 6.96m$ and $N = 14.4m$.

The simulation is run over thousands of timesteps, with several SIMPLE iterations in each timestep. The total wall-time of the simulations reaches several days, thus the linear solvers are rigorously tested. Initially, the numerics of the simulation are deliberately chosen to be sub-optimal, giving a poor flow solution. This highlights any numerical differences between the linear solvers, which are indeed significant.

For the Krylov solver and chaotic-cycle multigrid, the simulation is then run with much stricter numerics – smaller timesteps, greater number of SIMPLE iterations, and stricter ILCT (almost two weeks total wall-time) – section 8.5. Here, the two solvers arrive at virtually identical solutions, thus addressing concern (1).

Performance testing is performed in section 8.6, in the same manner as the scalability tests in previous chapters. Both points (2) and (3) are satisfied by these tests.

It should be clearly noted that this chapter does not aim to investigate the quality of the CFD simulation, the resolved flow, or any physical properties. Practically, even the largest mesh is too coarse to simulate this highly-unsteady test case accurately. The purpose of comparing flow solutions is purely to observe differences between the linear-solver configurations.

Furthermore, the main purpose of the performance testing in section 8.6 is not to measure scalability. The main purpose is to verify $O(N)$ scaling of the chaotic-cycle multigrid method, and investigate any differences in absolute performance due to the unsteady terms in the equation-system. Indeed, scalability bottlenecks are significantly reduced on the larger grids, because it is not possible to reduce the cells-per-core ratio to critical levels on Iridis4.

Overall, this chapter aims to ask: "are chaotic methods *suitable* for real-world CFD?", complementing the previous chapters which have focused on the *performance* and theoretical applicability of these methods.

8.1 Numerical Setup

The test case used for these studies is the Joubert-hull ("BB2") model-scale submarine [97], operating at a Reynolds number of approximately 5.2×10^6. This test case is used as a common benchmark for assessing the capabilities of CFD in submarine hydrodynamics (see, for example, the Submarine Hydrodynamics Working Group [86]). The length L of the vessel is $3.8260m$ and the flow velocity V is $1.6309m/s$. A yaw angle of $20°$ is applied by modifying the inflow vector, creating a flow which mimics a sharp left-hand turn. The grid is a multi-block structured mesh, as illustrated in figure 8.1. The forward and port-side boundaries are configured as inflow boundaries; with outflow boundaries opposite. The upper and lower boundaries are constrained by a pressure boundary condition. All hull surfaces are non-slip walls with fully-resolved boundary layers (no wall functions). The domain extends $2.5L$ forward of the hull, and $3.7L$ aft. The total height and width of the domain is $6L$. The origin, for the purpose of calculating moments on the hull, is at the aft of the vessel.

To begin, the simulation is deliberately set up as a 'worst case' in terms of numerical convergence, in order to amplify any possible issues arising from the use of non-deterministic, chaotic solvers. It is anticipated that with strict convergence, any problems would disappear (and this will be shown in section 8.5).

The inner-loop convergence tolerance (ILCT), for all equations, is set to 0.1. Large (implicit) timesteps are used (0.02 seconds, or $100 \times L/V$), and only 5 outer-loop (SIMPLE iterations) are performed per timestep. The average Courant number is 60, reaching 6000 in some boundary-layer elements.

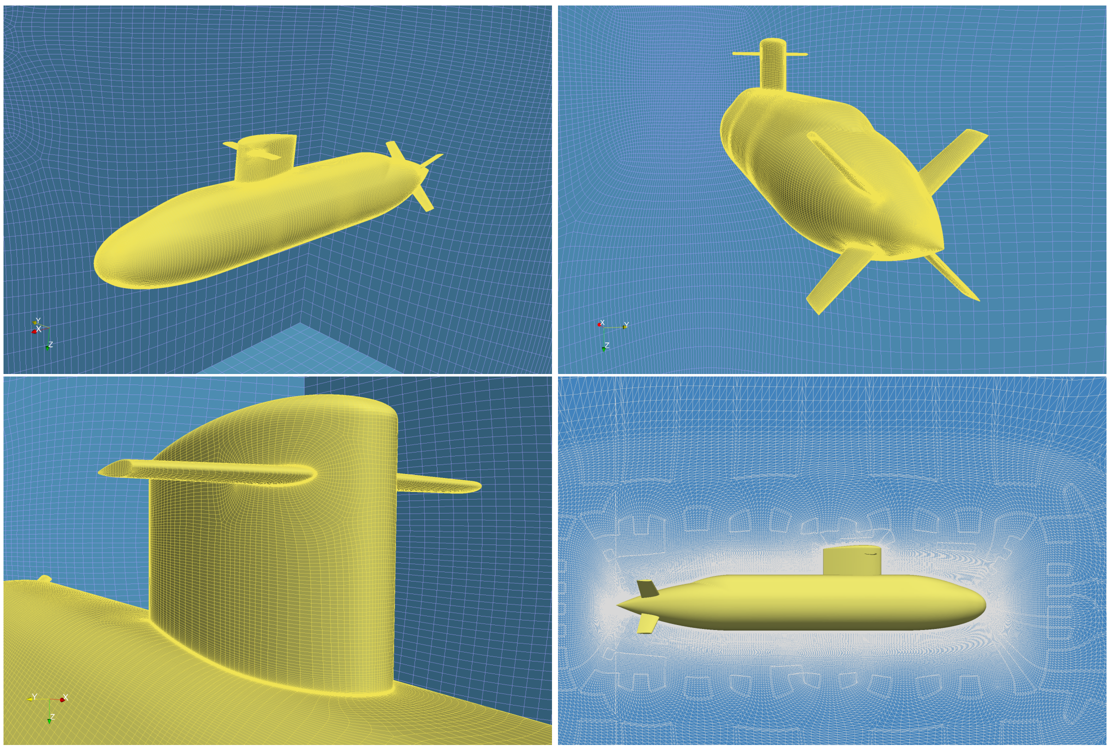


Figure 8.1: A view of the multi-block structured grid and domain for the BB2 test case, with $N = 14.4m$.

QUICK with limiters [44] is used for convective flux discretization of the momentum equation, whilst a first-order upwind method is used for turbulence. Turbulence is modelled with a two-equation k-ω shear-stress transport model [63]. Relaxation is applied to the outer-loop: $\alpha_{imp} = 0.9$ and $\alpha_{exp} = 0.15$ for momentum and turbulence; $\alpha_{exp} = 0.15$ for the pressure equation.

The total length of the simulation is 60 seconds, or 3000 timesteps. For the first 2000 timesteps Flexible GMRES is used with a Block Jacobi preconditioner – for all equations. The solution is then restarted with alternative pressure-equation solvers for the remaining 1000 timesteps. The turbulence and momentum equations are not altered[1]. 96 cores are used throughout – this is largely irrelevant, but does affect the partitioning of the multigrid algorithm/aggregation and the amount of asynchronicity occurring across MPI boundaries. For V-cycle and chaotic-cycle multigrid, re-aggregation occurs once per timestep; thus re-aggregation occurs 20× more frequently than in the previous chapter. This is done for safety, allowing re-aggregation to occur whenever the matrix coefficients change significantly, but is absolutely not necessary – further work would be required to find an optimum re-aggregation frequency[2].

[1]Previous results using chaotic methods for momentum and turbulence were troublesome, due to the ability of the chaotic methods to significantly over-converge (by 1-2 orders of magnitude) these equations between residual updates. This had a large (positive) effect on the overall flow residuals, but interfered with the most significant results regarding the expensive, elliptic pressure equation. The pressure equation is not noticeably over-converged by chaotic methods, simply because the time taken to reach convergence is much larger.

[2]A poor aggregation will never prevent convergence completely, since the multigrid method can never perform worse than the equivalent one-level solver. However, re-aggregating frequently ensures the full advantage of the multigrid method is maintained. As the solution converges, and matrix coefficients

In the following sections, the numerical results of these simulations are examined in order to determine the suitability of chaotic methods for CFD. Firstly, the convergence characteristics are examined. Following this, various aspects of the flow solution itself are observed – including quantitative results of friction/pressure distributions, and qualitative visualizations of the flow.

8.2 Convergence

Figure 8.2 shows the L_2-norm of the residual of the three momentum equations, two turbulence equations and pressure equation – on all three grids. Note that this is the residual of the SIMPLE, outer-loops – although it is closely related to the residual of the linear system. These residuals are computed by taking the *absolute* residual of each linear equation-system in each outer-loop, normalized by the diagonal of $\mathbf{A}$. In general, the residuals are quite unremarkable; but there is some variation between the linear-solver configurations. It is worth noting that the solutions ($\mathbf{x}$) provided by the linear solvers are allowed to be different, so long as the *relative* inner-loop convergence tolerance (ILCT) of the residual ($|\mathbf{r}|_2$) is reached (this was discussed earlier, in section 6.5). Perhaps this would be an argument for using a different check for inner-loop convergence, such as an absolute residual or an ∞-norm. However, when these residuals are visualized, by plotting regions of the flow which are poorly converged, there is no noticeable difference between the solvers – figure 8.4 shows this visualization.

Figure 8.3 shows the L_∞-norm of the same residuals. The differences between the solvers are greater (than when comparing L_2-norms), and also become larger as N increases, which is to be expected – the relative contributions (to the L_2-norm) of a few stagnated/under-converged equations becomes smaller as N increases, thus allowing a higher L_∞ and noisier results. It is interesting that the choice of linear solver for the solution of the linearized pressure equation affects the other transport equations more than the pressure equation itself, with particularly poor convergence of the momentum equations when using the FGMRES pressure-solver. It would be presumptive to draw any conclusions from this result, but it does appear that the non-Krylov solvers are more robust in this poor-convergence scenario. It is suspected that the multigrid and chaotic methods provide a smoother solution vector (due to superior high-frequency error smoothing), thus minimizing anomalous under-converged cells. Another argument could be made for checking ILCT (linear-system convergence) using an ∞-norm – reducing this problem and also removing dependence on N.

8.3 Transient Forces and Moments

The differences in the overall flow residual, when using various linear solvers, are quite large. This also translates to small differences in the physical properties of the flow. Figure 8.5 shows the non-dimensional total force on the hull; figure 8.6 shows total

stop changing significantly, there is likely to be a performance benefit to reducing the re-aggregation frequency.

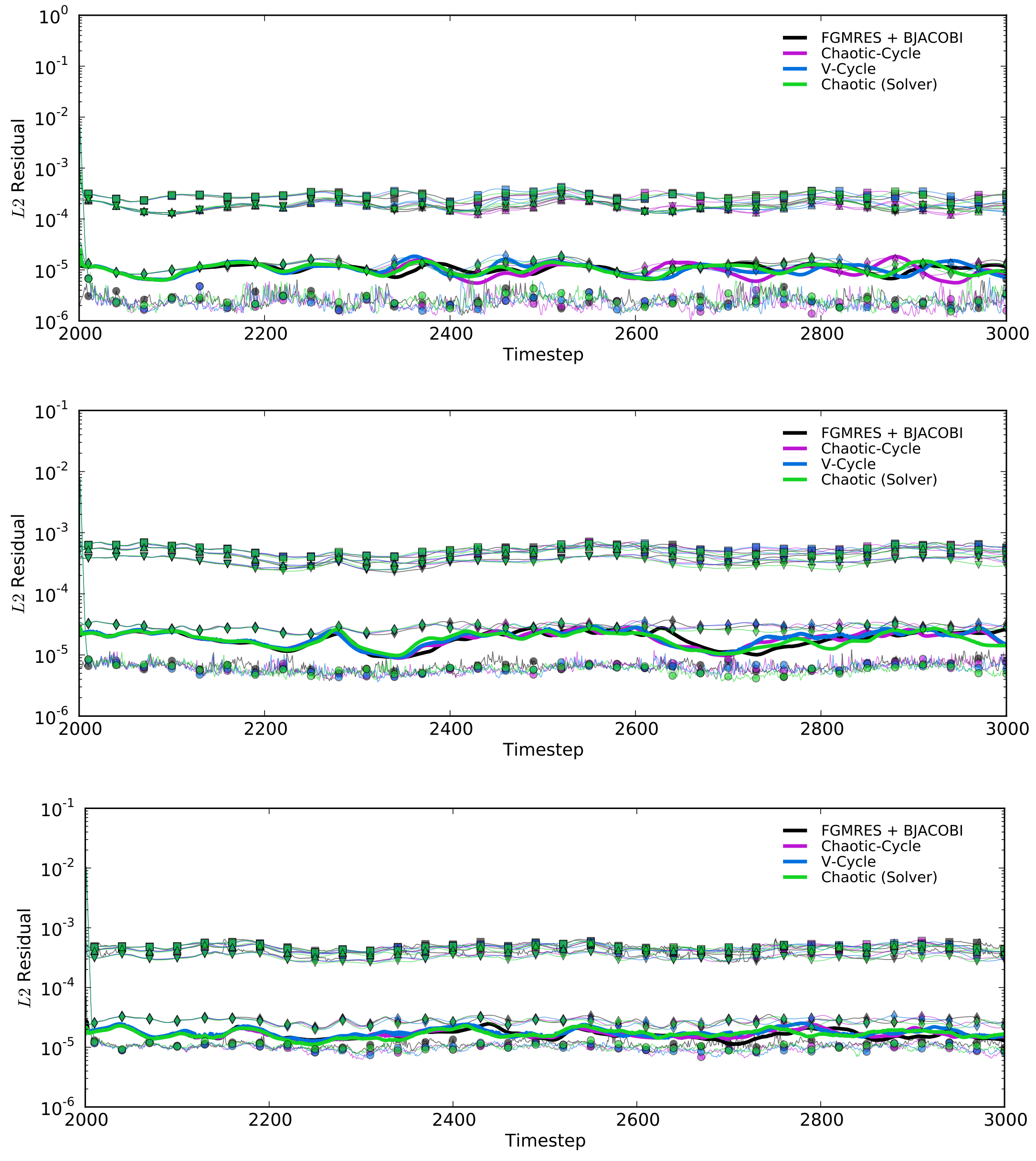


Figure 8.2: L_2-norm of the flow solution residuals, on three grids: $2.78m$ (top), $6.96m$ (middle) and $14.4m$ (bottom). The pressure equation is denoted with a thick solid line; momentum equations are shown with $x = ■$, $y = ▲$ and $z = ▼$; and turbulence equations are denoted with $k = ♦$ and $\omega = •$. The momentum and turbulence equations are translucent to improve clarity.

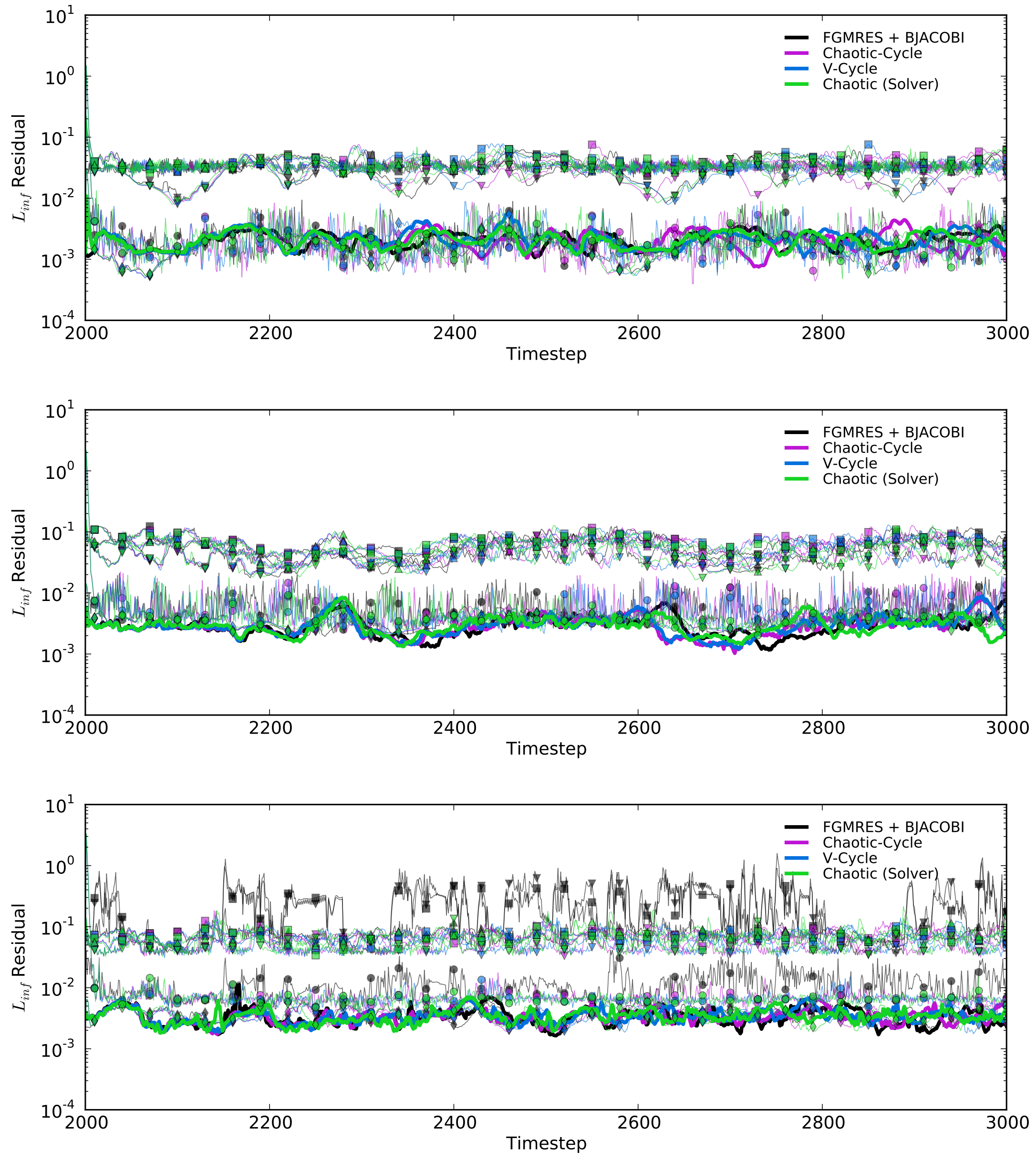


Figure 8.3: L_∞-norm of the flow solution residual, on three grids: $2.78m$ (top), $6.96m$ (middle) and $14.4m$ (bottom). The pressure equation is denoted with a thick solid line; momentum equations are shown with $x = ■$, $y = ▲$ and $z = ▼$; and turbulence equations are denoted with $k = ⧫$ and $\omega = •$. The momentum and turbulence equations are translucent to improve clarity. Notice the under-convergence of the x- and z-momentum equations (■, ▼) when using FGMRES (black) on the finest grid. This is the only significant difference between the residuals achieved by each linear solver configuration.

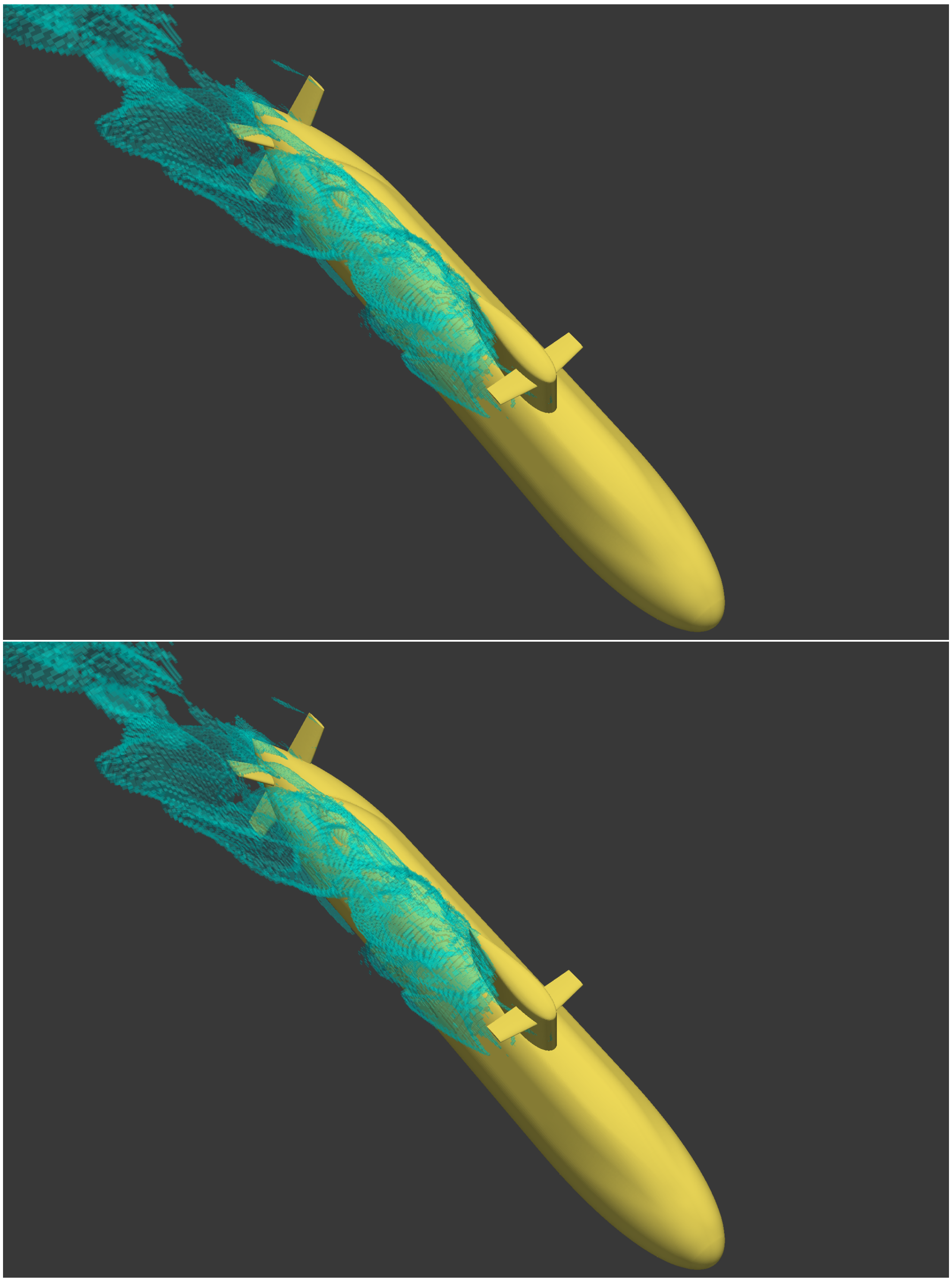


Figure 8.4: Control volumes in which the root-mean square of momentum-equation residuals ($\sqrt{L_x^2 + L_y^2 + L_z^2}$) is greater than 10^{-3}, showing poor convergence in the sail-wake region. Identical results were obtained for all linear solver configurations. FGMRES (top) and Chaotic-Cycle multigrid (bottom) are shown at the final timestep.

moments. Forces are non-dimensionalized with $\frac{1}{2}\rho V^2 L^2$; moments with $\frac{1}{2}\rho V^2 L^3$. Only the finest grid is shown ($N = 14.4m$) – the coarser grids provided an almost steady-state result for X, Y, Z, K, M and N with no perceptible difference between the various linear solvers.

It is not clear whether the variation of the forces and moments is from an attempt to capture the unsteadiness in the flow, or is an artifact of the poor convergence. To an extent, it appears that each solver provides a similar solution path, but with some amount of 'lag' – this is particularly visible in the heave force (Y). This suggests that some aspects of physical flow oscillation are captured, but is completely dominated by the poor numerics of the simulation. These plots will be revisited in section 8.5, where stricter convergence is used.

It is difficult to draw conclusions on which solver provides the most numerically-correct solution, but it is encouraging to see that the chaotic-cycle multigrid performs similarly to the V-cycle multigrid solver. This implies that the non-deterministic, chaotic nature is not having any negative impact on the flow solution.

8.4 Instantaneous Flow Solutions

More insight into the quality of the solution of each linear solver may be gained by observing local flow features at a single timestep. The solution of the finest grid is compared by measuring coefficients of friction and pressure at various slices along the hull; and, qualitatively, by taking slices of the fluid flow at interesting locations. As shown in the previous section, the transient behaviour of the flow and the numerical 'lag' leaves the simulation in a slightly different state, so for each solver the data has been extracted at a different timestep. The timestep was chosen by taking the last peak in Y force – this was both the largest and most periodic force experienced by the submarine, but any signal could be used with similar results. Thus the timesteps used were FGMRES at $t = 2950$, Chaotic-cycle at $t = 2905$, V-cycle at $t = 2890$ and Chaotic (solver) at $t = 2840$ (the penultimate peak, since it lags the least).

The coefficients of friction and pressure are shown in figures 8.7 and 8.8 respectively. Four slices are taken, at $x/L = 0.2, 0.4, 0.6, 0.8$ (from aft to bow). $x/L = 0.6$ includes the sail/conning tower, just aft of the two forward fins. $-180°$ corresponds to the port-most point of the vessel, whilst $-90°$ points directly upwards ($0°$ starboard, $90°$ downwards). The flow is generally equal, except in the lee/wake of the sail – this is the area in which convergence was worst, as shown in the previous section, figure 8.4.

Using the same timesteps, one can observe qualitative differences between the flow solution in other areas, when comparing the different solver configurations. Figure 8.9 shows surface limiting friction lines and coefficient of pressure on the hull surface, figure 8.10 shows velocity magnitude in the x-direction at a transverse slice ($x/L = 0.6$), and figure 8.11 shows the same at $y = 0$. The slices are relatively uninteresting, with no significant differences between the solvers observed. However, the surface friction lines show much more variation, particularly in the sail-wake area. It appears that most of the differences between the solvers are occuring in the highly-unsteady boundary layer

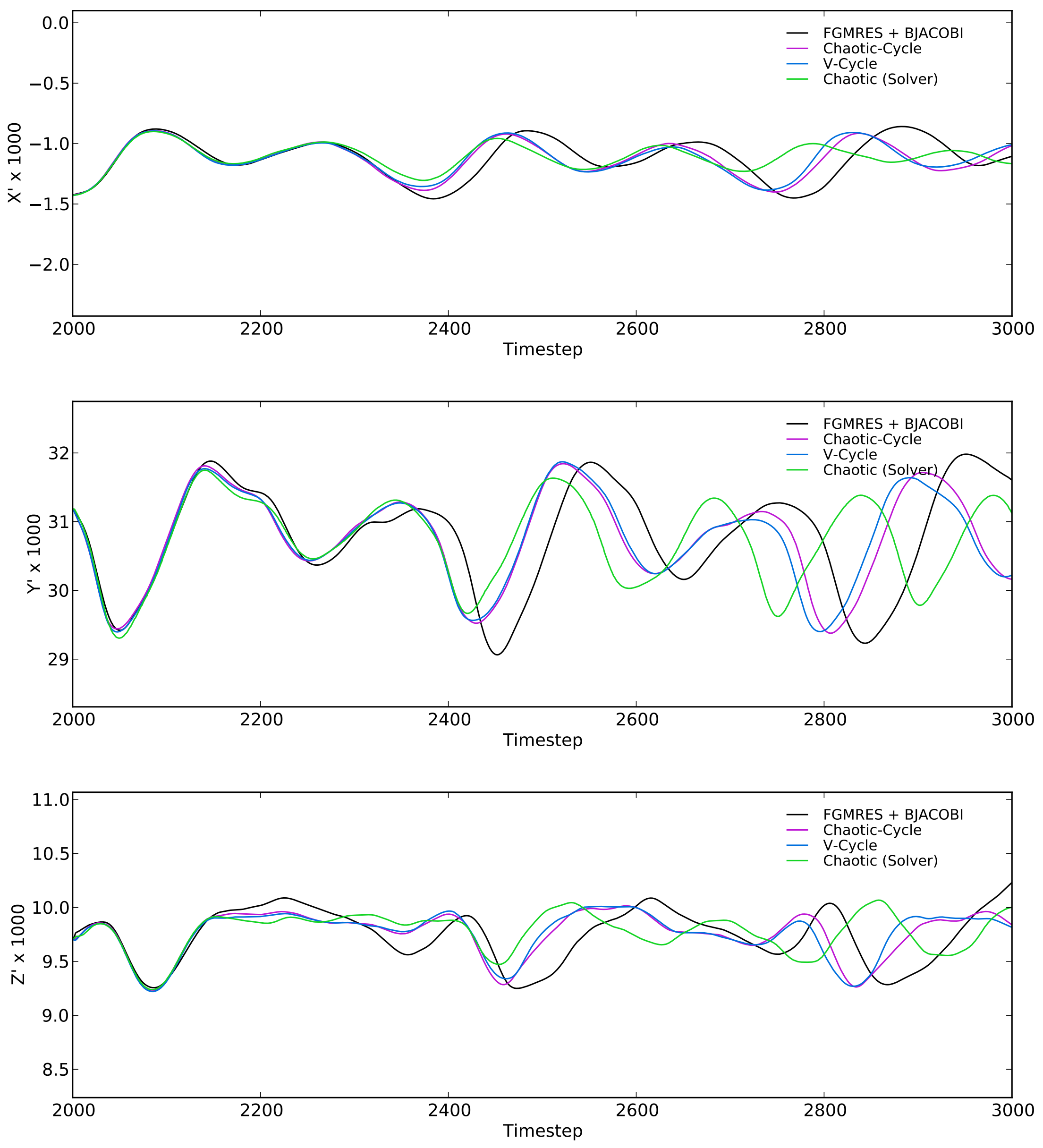


Figure 8.5: Non-dimensional total forces on the submarine hull in the x-, y- and z-dimension. Showing only the finest grid ($N = 14.4m$).

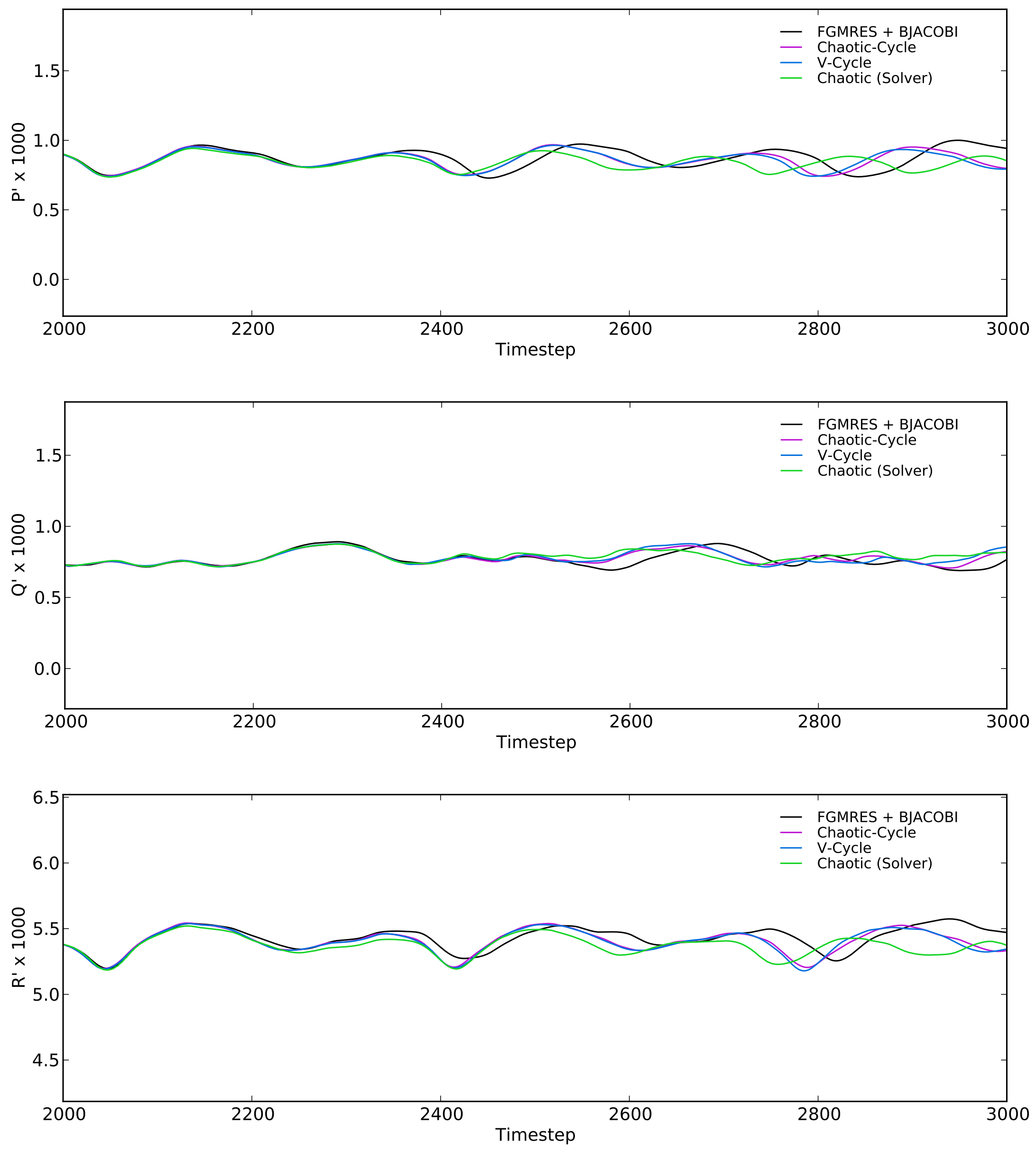


Figure 8.6: Non-dimensional total moments on the submarine hull around the x- y- and z-axis (P, Q and R respectively). Showing only the finest grid ($N = 14.4m$).

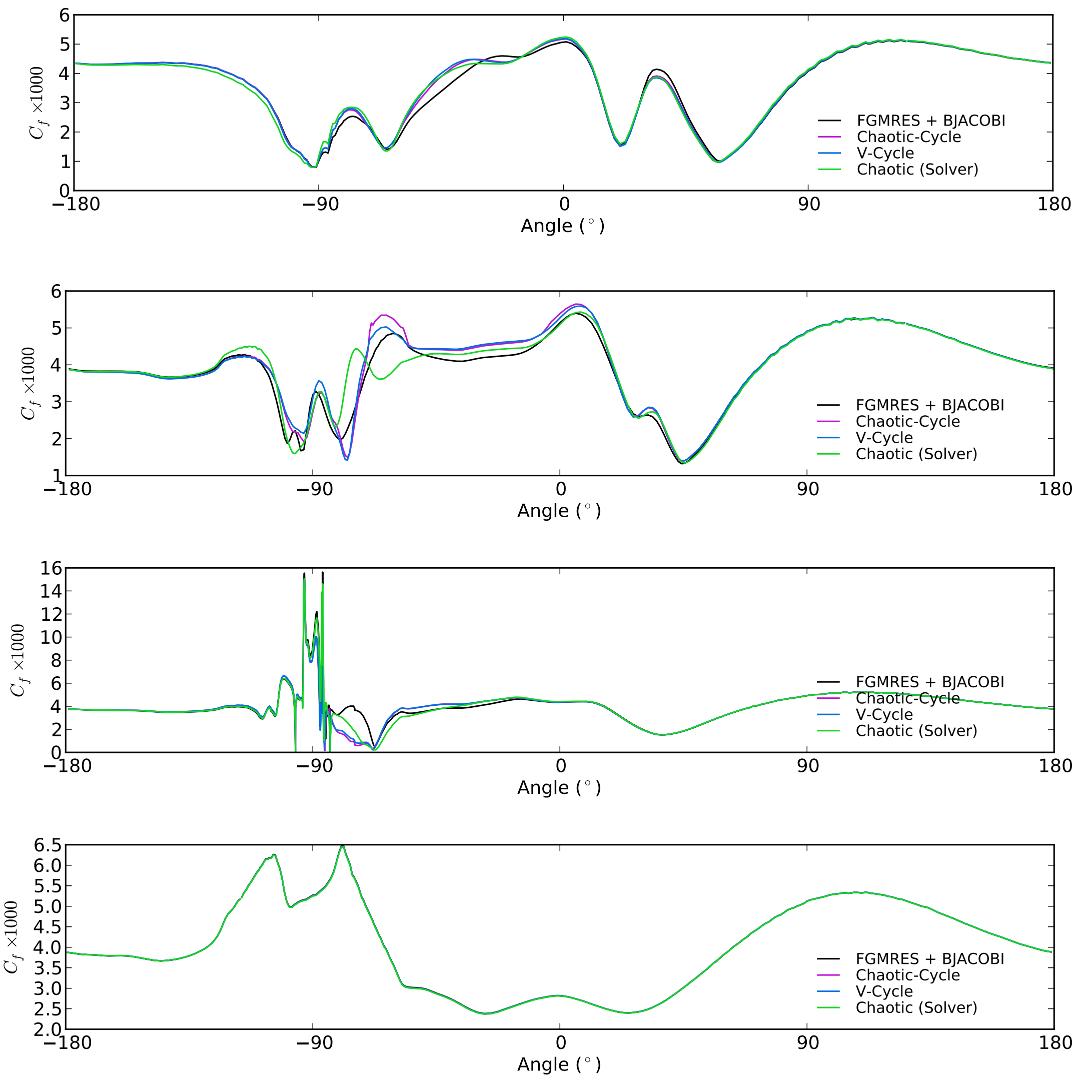


Figure 8.7: Coefficient of friction at 4 slices, from aft to bow at $x/L = 0.2, 0.4, 0.6, 0.8$ (top to bottom).

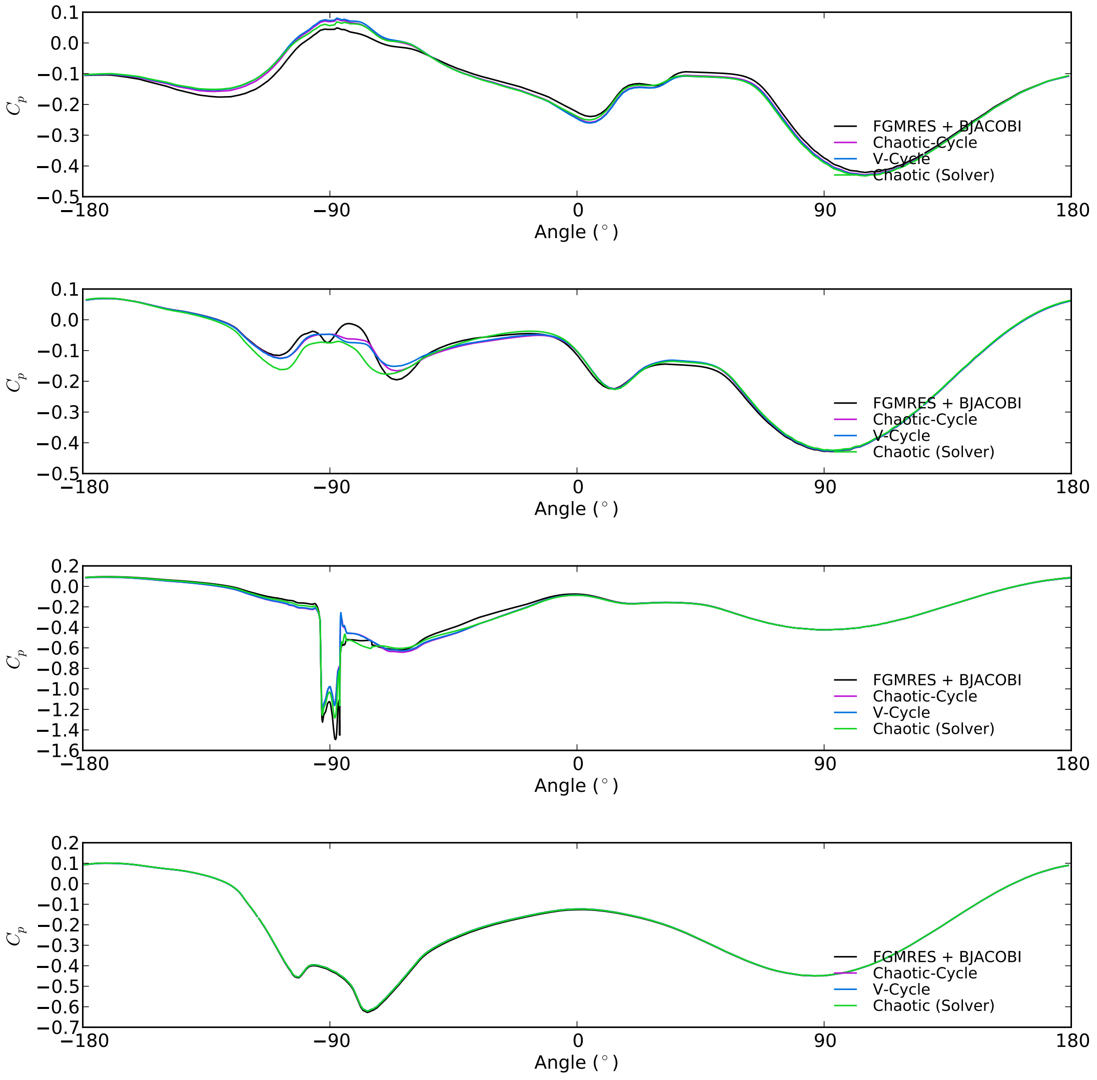


Figure 8.8: Coefficient of pressure at 4 slices, from aft to bow at $x/L = 0.2, 0.4, 0.6, 0.8$ (top to bottom).

in this region – where courant number and residuals are highest.

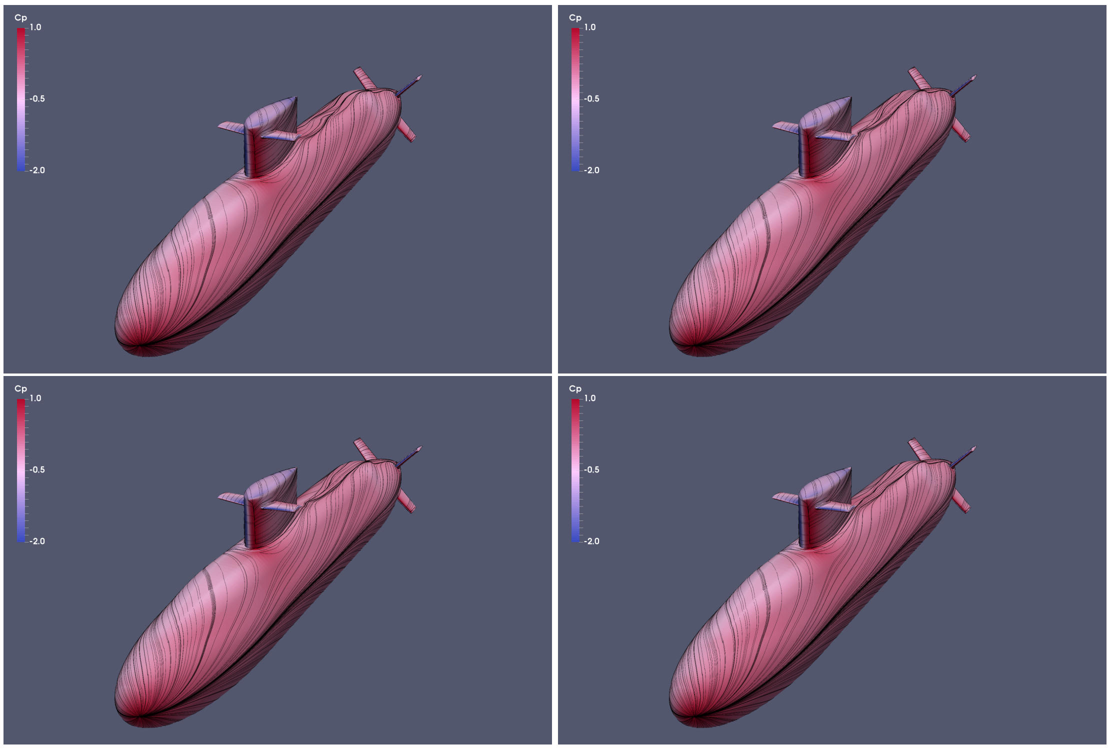


Figure 8.9: A visualization of the surface flow, showing limiting friction lines on the hull surface. The hull is also coloured with coefficient of pressure. The four solver configurations are shown: FGMRES + BJACOBI (top left), Chaotic-Cycle Multigrid (top right), V-Cycle Multigrid (bottom left) and plain chaotic solver (bottom right).

8.5 Stricter Convergence

The next logical to step is to determine whether the different pressure-solvers converge on exactly the same solution when overall convergence is improved and timestep fidelity is increased. To perform these tests, the solution is restarted after 1000 timesteps (20 seconds) of the original simulation. The timestep for the next 20 seconds is halved, to 0.01 (2000 timesteps). Finally, the last 20 seconds are covered using a timestep of 0.002 ($10,000$ timesteps). At the first restart, ILCT is also reduced to 0.01 (previously 0.1) and the number of SIMPLE iterations per timestep is increased to 15 (previously 5). It was not possible to run all solvers due to time constraints, but FGMRES and the chaotic-cycle multigrid were compared in this configuration – switching solver configuration at the first restart (20 seconds). The average Courant number dropped, as expected, to an average of approximately 0.6 – with a maximum of 69 in the boundary layer.

Overall, the L_2 and L_∞ residuals dropped by approximately an order of magnitude, compared to section 8.2, and the discrepancies between the two solvers are less apparent – see figure 8.12.

Figures 8.13 and 8.14 show the transient forces and moments respectively, for the last 20 seconds of the simulation. The two solvers show virtually identical results. This

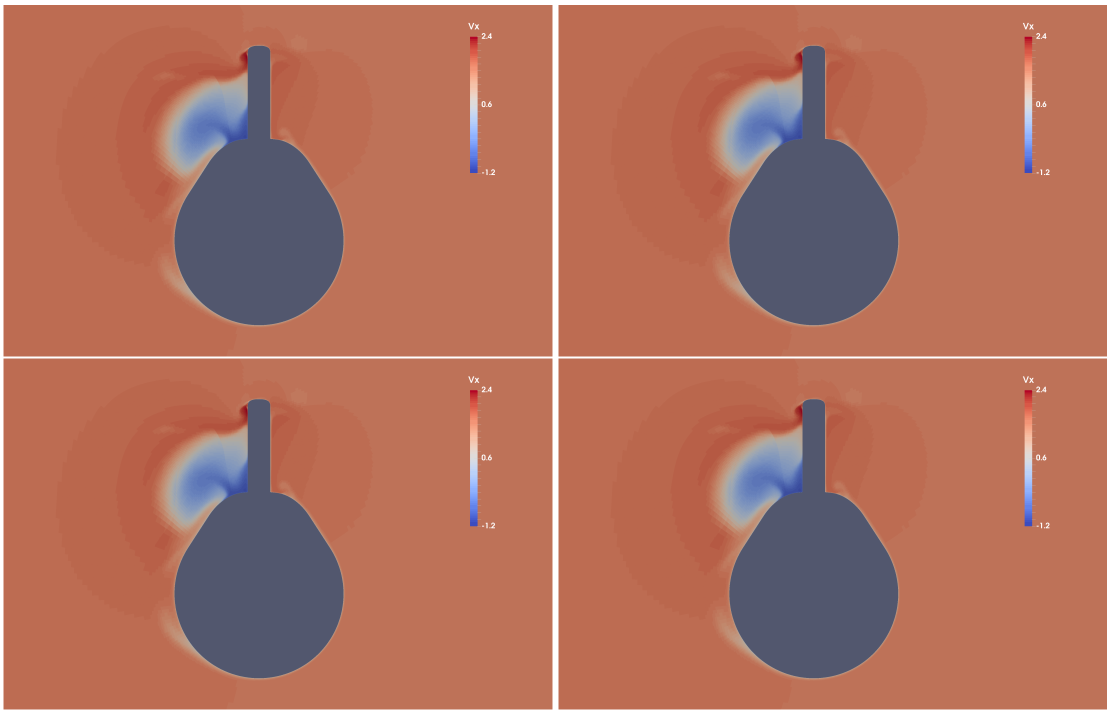


Figure 8.10: A slice of the simulation domain at $x/L = 0.6$, showing the magnitude of velocity in the x-direction. The four solver configurations are shown: FGMRES + BJACOBI (top left), Chaotic-Cycle Multigrid (top right), V-Cycle Multigrid (bottom left) and plain chaotic solver (bottom right).

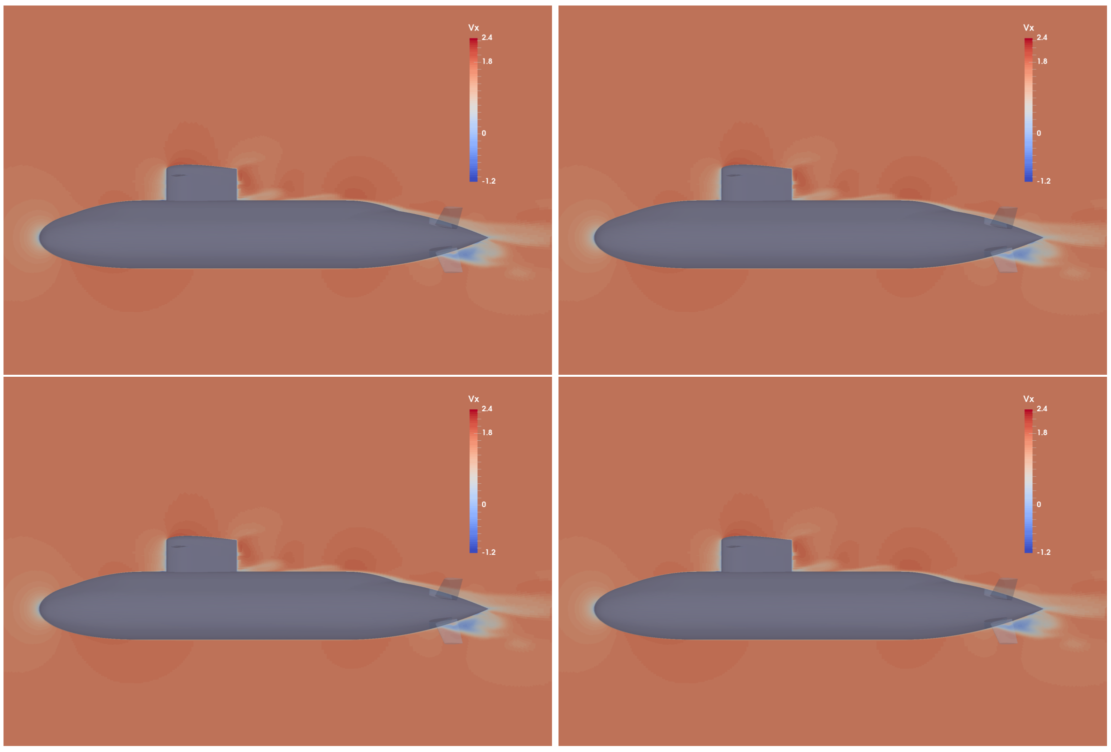
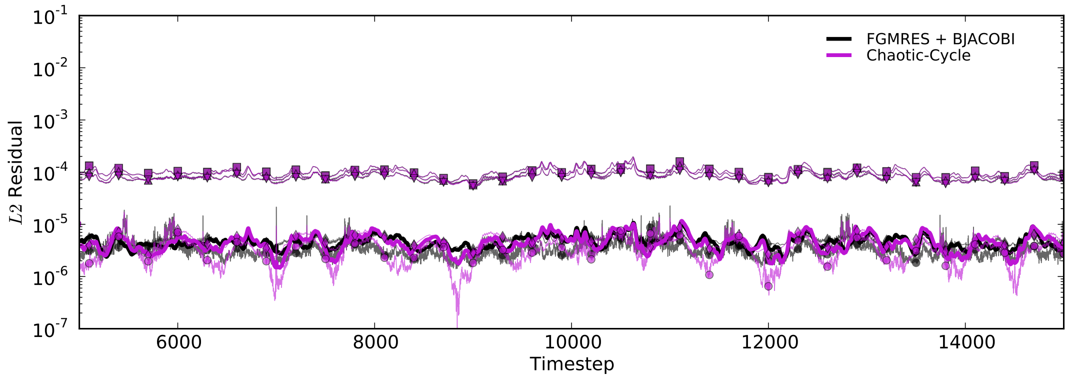


Figure 8.11: A slice of the simulation domain at $y = 0.6$, showing the magnitude of velocity in the x-direction. The four solver configurations are shown: FGMRES + BJACOBI (top left), Chaotic-Cycle Multigrid (top right), V-Cycle Multigrid (bottom left) and plain chaotic solver (bottom right).

is encouraging in terms of verifying the chaotic methods, but the numerical properties of the flow are still very poor. Proper periodicity or statistical convergence it not achieved, and would probably require a larger grid. Indeed, this type of flow can only be performed properly using a scale-resolving simulation (such as LES) – but it is beyond the scope of this thesis to investigate further, and not the purpose of this chapter.

Finally, figure 8.15 shows the limiting friction lines and coefficient of pressure, when comparing the two linear solvers. To all intents, the flow solutions are now equal, suggesting that there is *no negative affect on the flow solution when using chaotic methods* – assuming that overall convergence is good. This addresses the first concern raised at the beginning of this chapter.



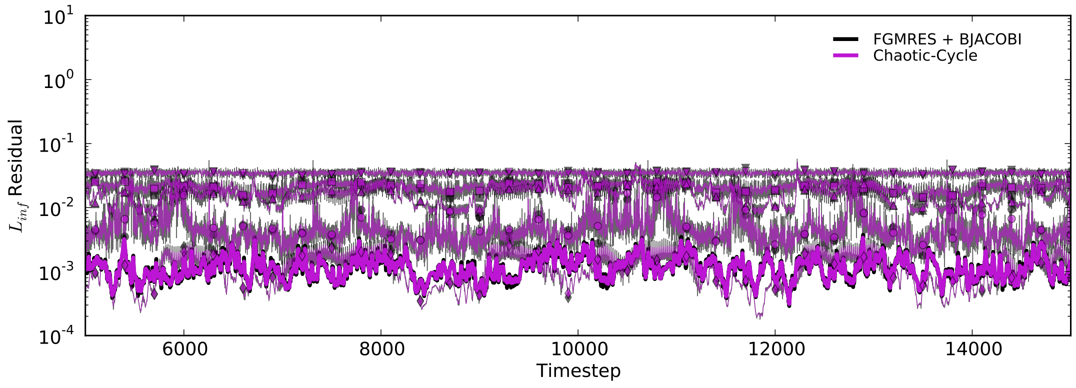


Figure 8.12: L_2-norm (top) and L_∞-norm (bottom) of the flow solution residuals, on the largest grid ($14.4m$ cells), using stricter convergence criteria. The pressure equation is denoted with a thick solid line; momentum equations are shown with $x = ■$, $y = ▲$ and $z = ▼$; and turbulence equations are denoted with $k = ◆$ and $\omega = •$. The momentum and turbulence equations are translucent to improve clarity. The last $10,000$ timesteps are shown, corresponding to the last 20 seconds.

8.6 Performance and Scalability of Chaotic Methods

Next, the scalability of each solver is measured – as in chapter 7. These scalability experiments are performed on each grid, over the first 100 iterations (20 timesteps) of the simulation. The performance is not expected to be significantly different to the KVLCC2 test case used in chapter 7; but the unsteady terms in the governing equations

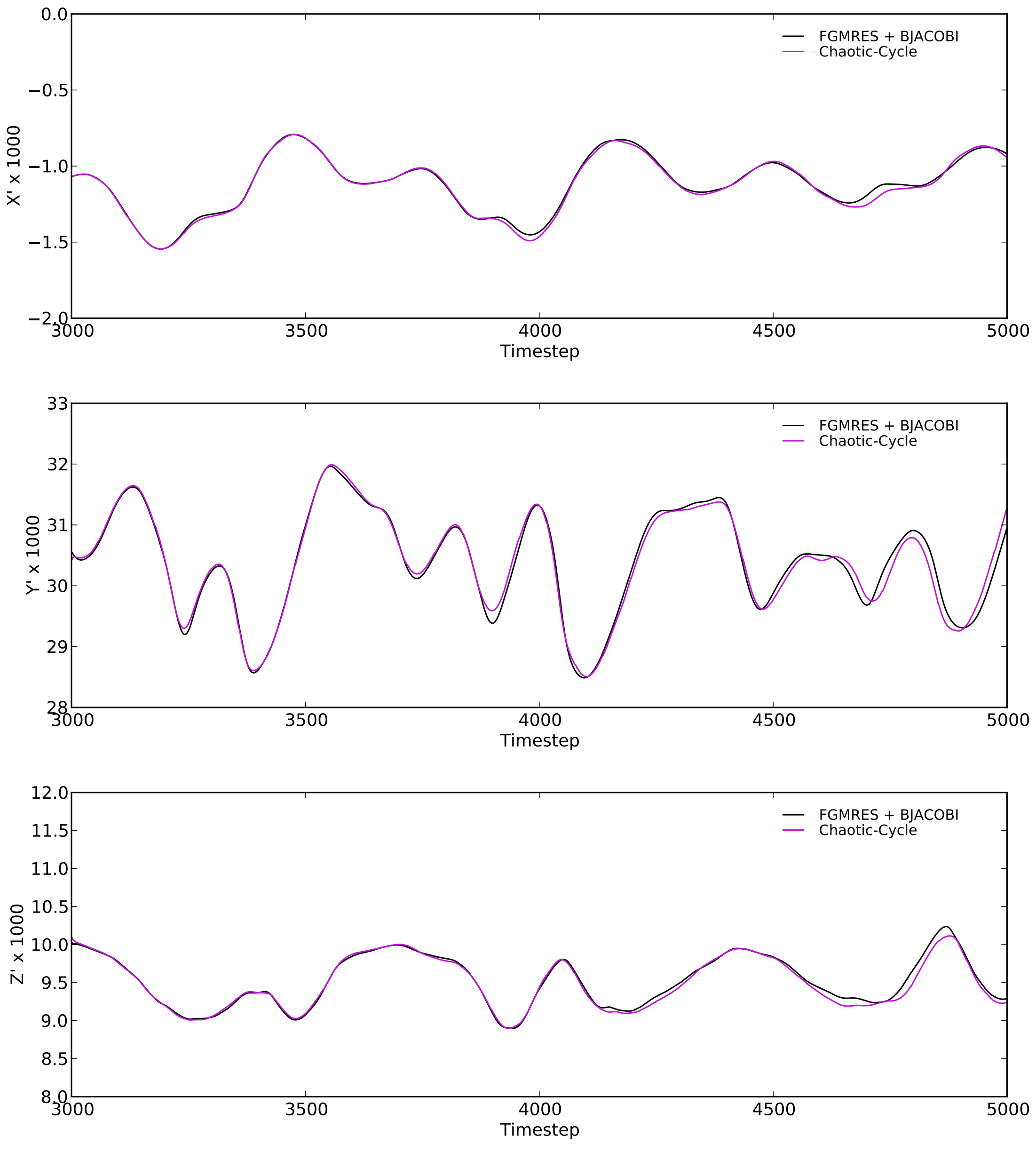


Figure 8.13: Non-dimensional total forces on the submarine hull in the x-, y- and z-dimension. Showing only the finest grid ($N = 14.4m$), with stricter convergence criteria.

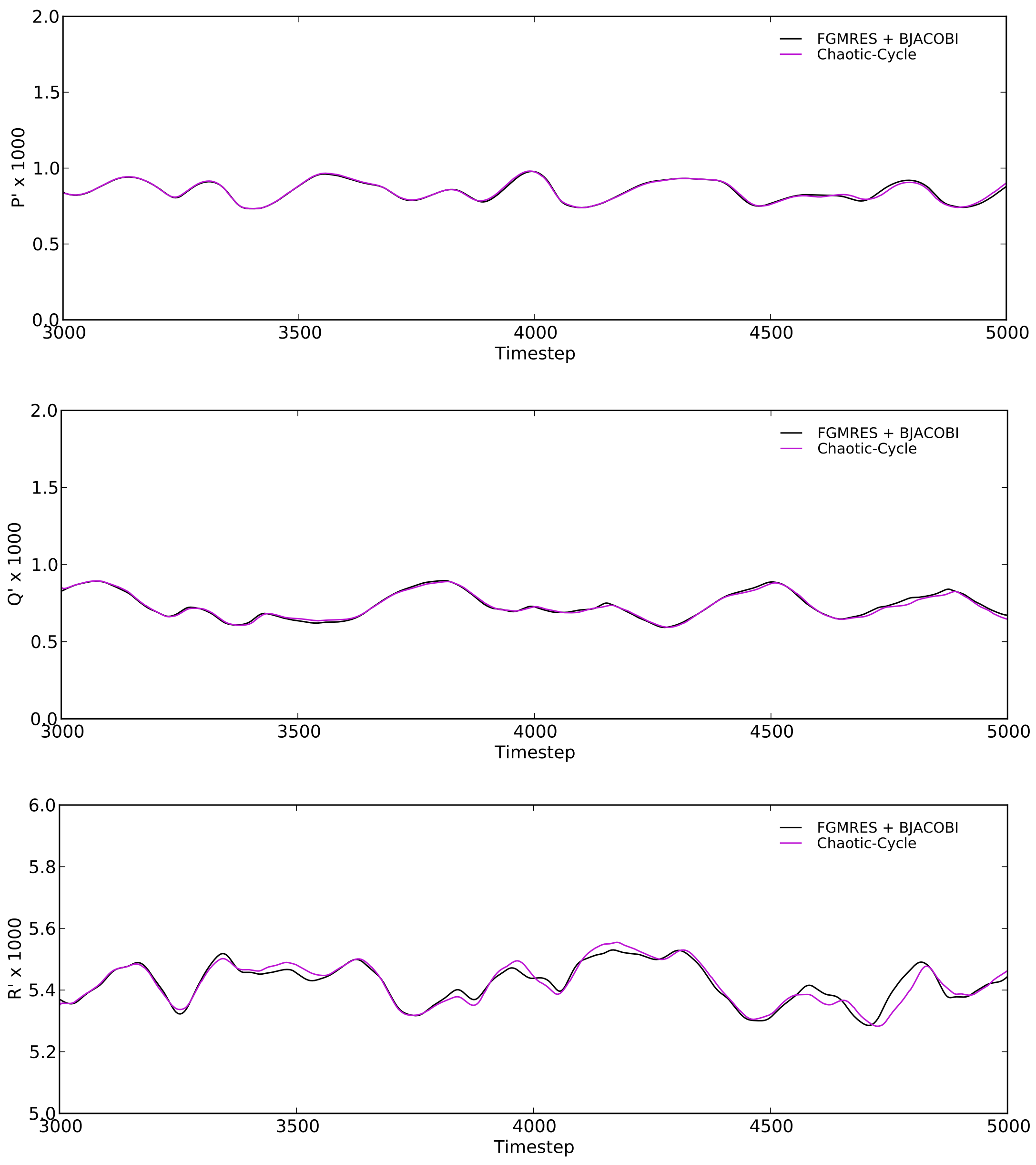


Figure 8.14: Non-dimensional total moments on the submarine hull around the x- y- and z-axis (P, Q and R respectively). Showing only the finest grid ($N = 14.4m$), with stricter convergence criteria.

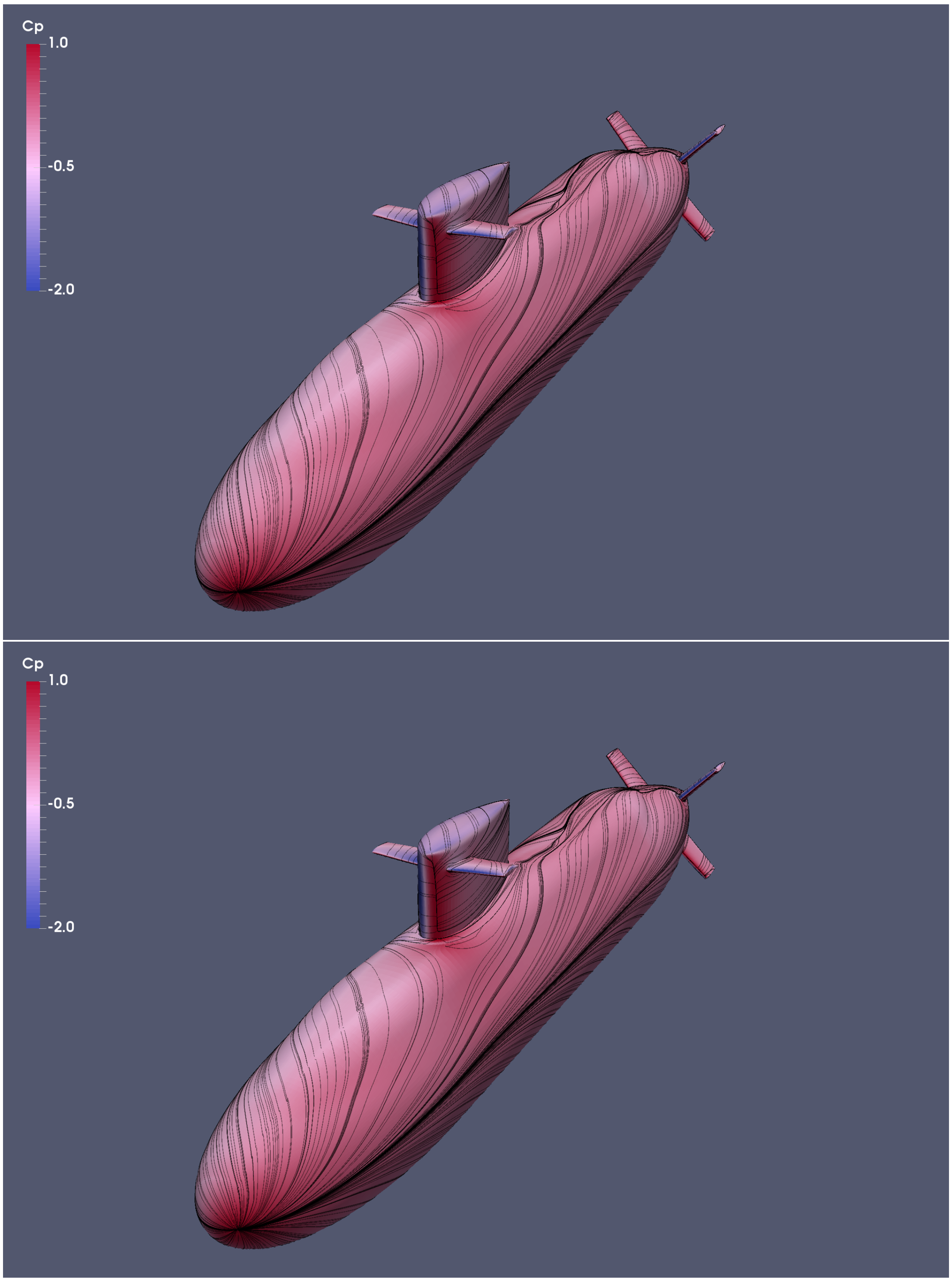


Figure 8.15: A visualization of the surface flow, showing limiting friction lines on the hull surface when using stricter convergence criteria. The hull is also coloured with coefficient of pressure. Two solver configurations are shown: FGMRES + BJACOBI (top) and Chaotic-Cycle Multigrid (bottom).

may assist the linear solver. The results are shown in figure 8.16 for ILCT of 0.1 and 0.01. It was, unfortunately, not possible to run these experiments on 1024/2048-cores[3], so care should be taken when comparing to figure 7.5. Only the time spent in the linear equation-system solver for the pressure equation is shown.

Scalability of the $N = 2.78m$ case is similar to results seen earlier for the KVLCC2. *No significant effects are observed due to the unsteadiness of the simulation*, although scalability of V-cycle and chaotic-cycle multigrid is improved due to extra time spent in the (very parallel-efficient) aggregation routines. This addresses the second concern raised at the beginning of this chapter.

As N increases, the cells-per-core ratio also increases, and the scalability bottlenecks of all solvers are reduced. Encouragingly, the performance of the multigrid methods (V-cycle and chaotic-cycle) remains competitive with FGMRES as N increases, suggesting that there are no issues applying these solvers to 'real-world' applications. However, within the realms of good scalability, FGMRES still outperforms these multigrid methods. If it were possible to increase the number of cores further, one would expect to see scalability issues, even on the largest grids, which would make the chaotic-cycle multigrid the fastest solver. Furthermore, reducing the relative amount of time spent in re-aggregation – by increasing the re-aggregation interval – would provide a significant benefit to both multigrid methods. On the largest grid, with $C = 512$, aggregation accounted for 13% of the total solution time of the chaotic-cycle multigrid.

Table 8.1 shows numeric results from these experiments, which helps to quantify the $O(N)$-scaling of the various solvers. None of the solvers achieve $O(N)$ performance according to this data but the differences between the various solvers are interesting. Both the chaotic-cycle and V-cycle perform similarly on a low number of cores (approximately $O(3N)$), with scalability differences making the comparisons on a larger number of cores more difficult. Since the chaotic-cycle scales better, and capitalizes on this when the cells-per-core ratio is lower, it's scaling with N appears worse. The N-scaling of the plain chaotic solver is worse, as expected, but is far from $O(N^3)$, due to the large amount of high-frequency errors which can be smoothed quickly (not spectral-radius limited).

Combining these results with the earlier verification of the chaotic-cycle multigrid (section 7.4), using a controlled Laplacian equation, this study addresses the third and final concern identified at the start of this chapter.

Table 8.1: Absolute core-hours spent solving the linearized pressure equation over 100 non-linear iterations on a variety of grid sizes with ILCT$= 0.01$. The final row shows the ratio difference between the finest and coarsest grid.

Grid Size	Chaotic (solver)			Chaotic-Cycle			V-Cycle			FGMRES		
	$C = 8$	128	512	8	128	512	8	128	512	8	128	512
2.78m	120.8	95.6	103.3	9.6	9.9	11.2	9.9	11.6	19.3	5.1	8.9	12.1
6.96m	475.3	475.2	375.1	30.0	38.1	33.8	32.5	56.5	54.3	18.7	28.9	36.6
14.4m	1482.4	1482.4	1319.5	139.1	114.6	98.1	110.2	168.9	123.8	50.7	87.3	87.5
×5.18	12.3	15.5	12.8	14.5	11.5	8.8	11.1	14.6	6.67	9.9	9.8	7.2

[3]These large jobs were only available during a rare, scheduled maintenance of Iridsi4

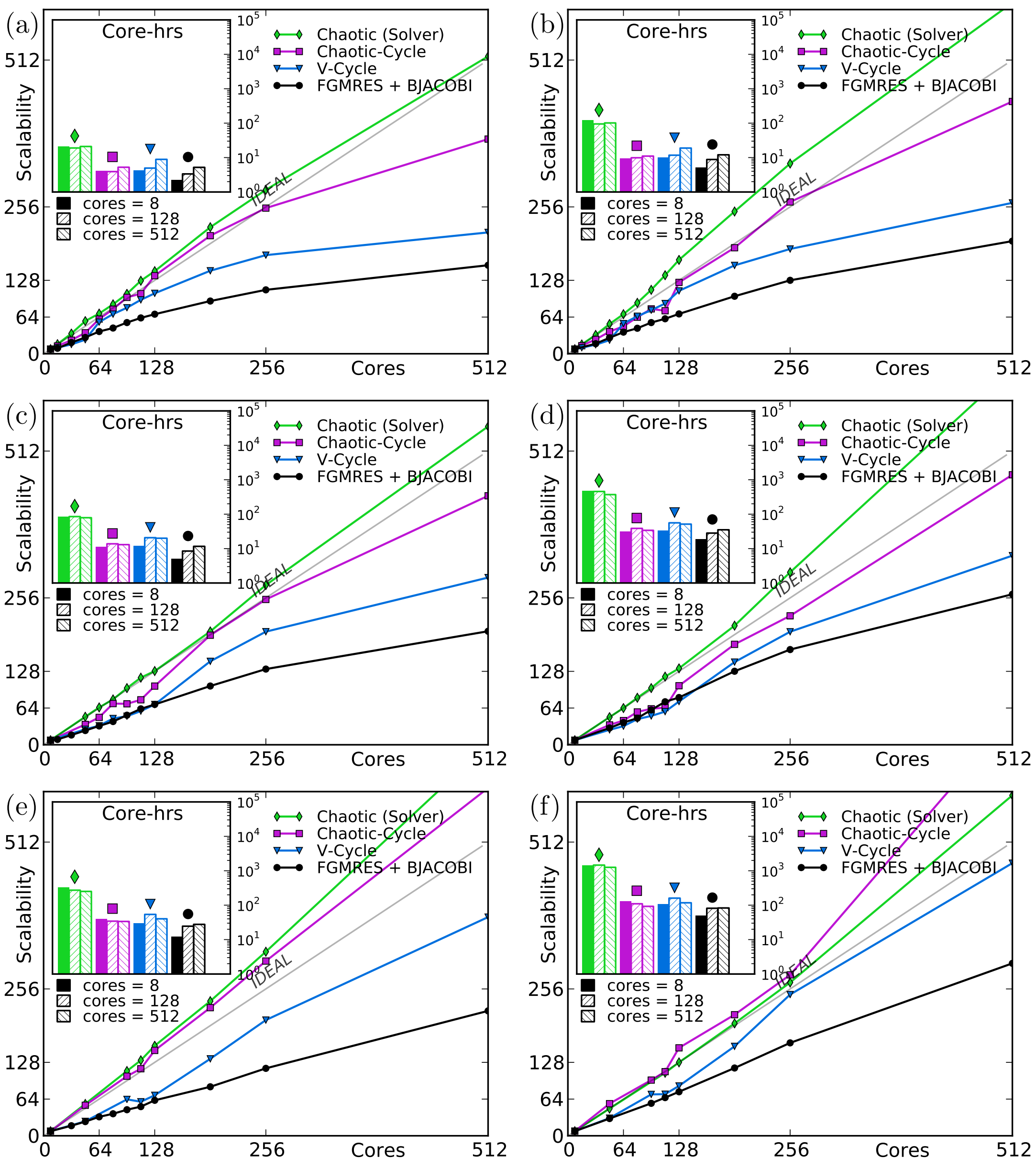


Figure 8.16: Scalability of the linear equation-system solver for the BB2 test case with $N = 2.78m$ (top), $N = 6.96m$ (middle) and $N = 14.4m$ (bottom) test cases, at ILCT= 0.1 (left) and 0.01 (right).

8.7 Conclusion

This chapter has answered some of the remaining uncertainties regarding the chaotic methods developed during this thesis. In particular, it has been shown that chaotic methods can be used safely in practical CFD simulations. Care should be taken when the overall flow solution is under-converged, however, because the solution will have a strong dependence on the linear solver that is used – but this occurs even when comparing classical solvers such as FGMRES and V-cycle multigrid, suggesting that this is unrelated to non-determinism.

This chapter has also shown that the performance of chaotic-cycle multigrid is maintained as N increases and the unsteadiness of the flow has no significant impact on the performance of the linear solvers.

The scalability of the chaotic-cycle multigrid is also good on the BB2 test case, even when N is large – as expected – but could not outperform FGMRES on the largest grids. As the number of cores increases to levels beyond which can be tested on Iridis4, the chaotic-cycle multigrid is expected to outperform these KSP methods.

Chapter 9

Conclusion

Exascale computing architectures and increasing dependence on unsteady simulations provided the motivation for the research question of this thesis:

> *"What limits the strong scalability of CFD and its ability to handle many-core architectures? What can be done to improve the CFD algorithms in this respect?"*

To answer in short: the limit to the strong scalability of CFD is the unscalable solve routines, used to solve the linear equation-system arising in the semi-implicit CFD algorithm – occupying up to 95% of total wall-time on 2048 cores. One means of removing this bottleneck is to use the theory of 'Chaotic Relaxation' to create synchronization-avoiding linear solvers which are highly scalable. A novel 'Chaotic-Cycle' multigrid method has been developed in this thesis – offering superior scalability, and speed-ups up to 13.3× compared to classical solvers (on 2048 cores). In principle, this would accelerate the CFD algorithm by a factor of 8.2× on 2048 cores (via Amdahl's Law [1]), marking a significant improvement to the strong scalability of CFD. The solver has been implemented, verified, tested and released as part of an open-source library (*Chaos*[1]) – for transparency and to facilitate further work.

The process of defining the research question began with a thorough investigation into the key technologies associated with high-performance computing and next-generation CFD (chapter 3). Concerning HPC, the trend towards many-core exascale machines highlighted key problems for scalability, especially related to communication bottlenecks, heterogeneity and memory bandwidth. For CFD, there are clear trends towards higher-fidelity modelling which create additional scalability problems, since time-stepping must occur sequentially and spatial dimensions must be more finely partitioned. These issues were the primary motivation for the research question. A number of other key technologies were identified at this stage, such as parallelization and acceleration schemes, multi-disciplinary coupling techniques, higher-order methods and automatic meshing – some of which had to be borne in mind when answering the research question.

[1] `https://bitbucket.org/jamesnhawkes/chaos`

An initial study was performed in order to identify the current bottlenecks to strong scalability (chapter 4), using ReFRESCO – a state-of-the-art semi-implicit CFD code. For the most part, common simulation parameters such as discretization scheme, mesh structure, and choice of turbulence models had no significant effect on scalability. However, there was a large dependence on the 'inner-loop convergence tolerance' (ILCT) – the tolerance to which linear sub-systems are solved – and the choice of linear sub-system solver. For the industry-standard, wind-tunnel KVLCC2 test case with an ILCT of 0.1, on 2048 cores, 85% of the solution time was spent in the linear solver, rising as ILCT decreased. Most of this was spent solving the elliptic pressure equation. Profiling showed that the parallel efficiency of the linear solvers was poor due to memory bandwidth contention on a single node; and expensive global communications as the number of cores increased. Other routines, such as matrix assembly and gradient calculations, scaled to 2048 cores with only minor issues.

Investigation into alternative linear solvers and preconditioners was conducted in chapter 5. These experiments were performed up to 512 cores, where the parallel efficiency of the *de facto* linear solver (Flexible GMRES with a Block Jacobi preconditioner) was between 13.6% and 25.1%. Good scalability was noted from stationary solvers, such as successive over-relaxation (SOR) – up to 64% parallel efficiency; but their absolute performance was weak, requiring up to an order of magnitude more wall-time. A hybrid method of FGMRES with 10 iterations of SOR as a preconditioner worked well, with 57% parallel efficiency; however, the absolute wall-time on 512 cores was barely better than with a Block Jacobi preconditioner when an ILCT of 0.01 was required – due to spectral-radius-limited convergence rates. A multigrid preconditioner was tested in these investigations, but it showed much worse scalability. This appeared to be because of synchronization costs in the smoothing routines, although it was difficult to profile in detail.

Following these unsatisfactory results, development began on linear solvers which would provide optimum scalability without sacrificing absolute performance (chapter 6). The starting point for this was the theory of 'chaotic relaxation', which provides a means of desynchronizing iterations in SOR-like solvers. A novel 'chaotic solver' was implemented, using a hybrid (MPI + OpenMP) paralellization scheme. Within each MPI process several 'computation' threads iterate through SOR-like relaxations, whilst a 'communication' thread is used solely for inter-process MPI communications. The threads operate completely out-of-synch, exploiting deliberate data-races to increase scalability and numerical performance.

The correctness of the chaotic solver was verified by examining the residual vector of the solution. Scalability tests were then performed. On 512 cores scalability was super-linear, due to effective use of the cache – beating the scalability of all solvers tested in the previous chapter. On 2048 cores, scalability worsened slightly, but overall parallel efficiency was still good; and reducing the interval at which residuals are computed would improve the scalability further. Absolute performance was up to $\times 37.8$ faster than plain SOR; and often out-performed FGMRES (almost $\times 10$ faster in some cases).

However, there are concerns relating to how chaotic or SOR-like solvers scale with the number of unknowns, N; because the stiffest error frequencies in the system cannot be solved faster than $O(N^3)$.

Following this, a method for combining the strong scalability of chaotic methods with the good numerical performance of multigrid methods was explored (chapter 7). A classical multigrid solver framework was created, utilizing unsmoothed aggregation and employing normal V-, W- and F-cycles. Extending this, a novel 'chaotic-cycle' multigrid method was created, using aspects of the chaotic solver and a classical sawtooth-cycle to remove implicit synchronization in the multigrid method. Verification of (close to) $O(N)$ performance was shown using a pragmatic Laplacian test case, and scalability tests were performed. The parallel efficiency of the chaotic-cycle multigrid was approximately 38% (on 2048 cores), compared to just 6% for FGMRES and 3% for V-cycle multigrid. The reasons for this less-then-ideal scalability have been identified: poor implementation of coarse-grid communications; and too-frequent residual checks. Correcting these issues should bring the chaotic-cycle multigrid method close to ideal or super-linear scalability. Importantly, in absolute terms, the chaotic-cycle multigrid is already up to 7.7$\times$ faster than Flexible-GMRES and 13.3$\times$ faster than classical V-cycle multigrid on 2048 cores.

The final content chapter of this thesis has taken the opportunity to test the suitability of the chaotic methods to realistic CFD simulations (chapter 8). Although care must be taken when the overall CFD solution is not properly converged, the results verified that the non-deterministic methods can be used safely for practical CFD and that they are entirely suitable to replace existing solvers. Furthermore, the experiments demonstrated (again) that the performance of the chaotic-cycle is maintained as N is increased, and showed that the unsteady terms in the CFD algorithm had no significant impact on linear solver performance.

Based on the scalability, performance and suitability of this chaotic-cycle multigrid solver, it is safe to conclude that this novel solver makes considerable steps towards improving the strong scalability of CFD. The chaotic-cycle solver could already accelerate ReFRESCO by a factor of 5.8$\times$ compared to KSP methods, or 8.2$\times$ compared to classical V-Cycle; and these gains are likely to increase as new hardware worsens the scalability bottlenecks of existing methods.

Moreover, there are no foreseeable issues associated with applying chaotic-cycle multigrid methods to other elliptic and non-elliptic, sparse, linear equation-systems, arising in other numerical sciences. To this end, all of the methods developed during this PhD, including the plain chaotic solver, classical multigrid and chaotic-cycle multigrid, have been implemented as an open-source C++ library, *Chaos*. This library should allow further developments of the chaotic methods and facilitate testing with other CFD codes, disciplines and numerical fields.

A vision for the future of *Chaos* is laid out in the following chapter, discussing the scope for further work – including improvements to the algorithm and implementation; better integration with MPI-only codes such as ReFRESCO; and variations on the chaotic methods already tested.

Chapter 10

Further Research

There are a myriad of variations and alternative implementations which could stem from the research into chaotic methods conducted in this thesis. However, the first objective should be to remove the scalability bottlenecks which have already been identified: the poor implementation of coarse-grid communications and too-frequent residual computations. In tandem with these improvements, investigations into one-sided MPI communications, allowing desynchronization of communication threads, could be fruitful.

To interface correctly with the hybrid *Chaos* library, ReFRESCO should adopt a hybrid paralellization scheme. This was the original intention and part of the reason for investigating hybrid solvers, but unfortunately this could not be achieved during this PhD. Perhaps a more versatile solution would be to adapt *Chaos* so that it can interface with MPI-only codes – not just ReFRESCO. A completely flexible interface is envisioned, where shared-memory multi-threading is used internally (keeping the concept of communication and computation threads) but MPI-only codes could interface using shared-memory windows. There are many recent developments in the MPI standard which may enhance this research [66]. This would also allow *Chaos* to be included in other established linear-solver packages such as PETSc.

With these improvements in place, much research can be performed on variations to the chaotic-cycle. For example, there may be benefits to re-introducing pre-smoothing (à la V-cycle), or varying the 'boundedness' of the chaotic smoother (by relaxing the constraints on the 'k' iteration counter). There may also be significant benefits to improving the aggregation process. Although the benefits of better aggregation would not be unique to chaotic-cycle multigrid, adaptation of the chaotic-cycle multigrid to smoothed aggregation operators may pose some unique challenges and opportunities. Additionally, some further research should be performed regarding re-aggregation frequencies – particularly for unsteady CFD simulations.

As mentioned, the numerical theory and practical implementation of the chaotic methods are closely coupled – and this is part of the reason for releasing *Chaos* as an open-source library. However, the underlying chaotic principles and novel algorithms should be quite hardware-agnostic. Indeed, chaotic methods should be well suited to heterogeneous hardware, including accelerated or mixed-core processors, because they mask the complexities of variable compute power, variable communication latency and load-balancing. In section 7.7, the possibility of running the chaotic methods on the Xeon Phi was demonstrated, but specializations of the code are probably needed in order to get the best results, and this will require further work.

Linking to the wider CFD context, there have been recent developments towards coupling of momentum and pressure equations in the SIMPLE loop [53]. The resultant linear equation system is a multi-block linear system, which could potentially be solved with chaotic methods. There have also been developments into higher-order discretization methods, such as discontinuous Galerkin methods [69], which do cause some unique problems for linear solvers and multigrid methods in general [72]. These are both areas which should be explored as the technology matures.

Due to the issues of interfacing *Chaos* and ReFRESCO, the original scalability studies could not be reproduced fairly (since the remainder of ReFRESCO is using $\frac{1}{8}^{th}$ of the available hardware), but it was possible to predict the effects of chaotic methods in the context of the entire CFD algorithm (approximately a $\times 5.8$ to $\times 8.2$ speed-up on 2048 cores). Unfortunately, predicting scalability to higher core-counts is almost impossible. In the SIMPLE algorithm itself, there are parts of the code which scale with cells-per-core ratio; parts which scale exclusively with the number of processors; and parts which scale unintuitively with local communications, memory bandwidth and cache use. With overlapping of communications and computations these interactions become more complex, and it becomes very difficult to predict scalability bottlenecks. Chaotic methods make this even more difficult. The only way to thoroughly assess the scalability of chaotic methods on many-core, exascale architectures will be to test them on these architectures.

For CFD as a whole, there are still many challenges which must be addressed in order to guarantee effective use of exascale supercomputers. Scalability issues relating to file I/O, adaptive meshing, overset meshing, effective pre- & post-processing and multi-disciplinary coupling are important areas to address. These areas are wide-ranging and will affect different codes and applications to varying degrees, whereas this thesis has tried to tackle the universal core of the CFD algorithm. It is hoped that the scalability studies and novel research presented in this thesis will pave the way to truly scalable CFD.

References

[1] Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, New York, NY, USA. doi:10.1145/1465482.1465560.

[2] Anzt, H., Tomov, S., Dongarra, J. & Heuveline, V. (2013, December). A Block-Asynchronous Relaxation Method for Graphics Processing Units. *Journal of Parallel and Distributed Computing*, 73(12):1613–1626.

[3] Anzt, H., Tomov, S., Gates, M., Dongarra, J. & Heuveline, V. (2012). Block-asynchronous Multigrid Smoothers for GPU-accelerated Systems. *Procedia Computer Science*, 9:7 – 16. ISSN 1877-0509. doi:http://dx.doi.org/10.1016/j.procs.2012.04.002. Proceedings of the International Conference on Computational Science, {ICCS} 2012.

[4] Argonne Leadership Computing Facility (ALCF), Illinois (Acc. 2015, May). `http://aurora.alcf.anl.gov/`.

[5] Bahi, J. (2000). Asynchronous Iterative Algorithms for Nonexpansive Linear Systems. *Journal of Parallel and Distributed Computing*, 60:92–112.

[6] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F. & Zhang, H. (2013). PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory. `http://www.mcs.anl.gov/petsc`.

[7] Bandringa, H., Verstappen, R., Wubbs, F., Klaij, C. & Ploeg, A. (2012, October). On Novel Simulation Methods for Complex Flows in Maritime Applications. *Numerical Towing Tank Symposium (NUTTS)*, Cortona, Italy.

[8] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. & der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA.

[9] Basermann, A., Kersken, H.-P., Schreiber, A., Gerhold, T., Jgerskpper, J., Kroll, N., Backhaus, J., Kgeler, E., Alrutz, T., Simmendinger, C., Feldhoff, K., Krzikalla,

O., Mller-Pfefferkorn, R., Puetz, M., Aumann, P., Knobloch, O., Hunger, J. & Zscherp, C. (2012). HICFD: Highly Efficient Implementation of CFD Codes for HPC Many-Core Architectures. In C. Bischof, H.-G. Hegering, W. E. Nagel & G. Wittum, editors, *Competence in High Performance Computing*, pages 1–13. Springer Berlin Heidelberg.

[10] Baudet, G. M. (1978, April). Asynchronous Iterative Methods for Multiprocessors. *J. ACM*, 25(2):226–244.

[11] Bhushan, S., Carrica, P., Yang, J. & Stern, F. (2011). Scalability Studies and Large Grid Computations for Surface Combatant Using CFDShip-Iowa. *IJHPCA*, 25(4):466–487.

[12] Browne, S., Dongarra, J., Garner, N., Ho, G. & Mucci, P. (2000, August). A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204. ISSN 1094-3420. doi: 10.1177/109434200001400303.

[13] CD-Adapco (2010, July). Star-CCM+ Performance Benchmark and Profiling. *HPC Advisory Council (Best Practices).*

[14] CD-adapco (2014). STAR-CCM+. `http://www.cd-adapco.com`.

[15] CD-Adapco (Acc. 2015, May). CD-Adapco and HLRS Push the Boundaries of HPC Performance with STAR-CCM+. `http://www.cd-adapco.com/pr/cd-adapco-and-hlrs-push-boundaries-hpc-performance-star-ccm`.

[16] Chazan, D. & Miranker, W. (1969, April). Chaotic Relaxation. *Linear Algebra and its Applications*, 2(2):199–222.

[17] Coteus, P. W., Knickerbocker, J. U., Lam, C. H. & Vlasov, Y. A. (2011). Technologies for Exascale Systems. *IBM J. Research and Development*, 55(5):1–12.

[18] Culpo, M. (2011). Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters. Technical report, Partnership for Advanced Computing in Europe.

[19] D. Mize (2012, February). Scalability of ANSYS Applications on Multi-Core and Floating Point Accelerator Processor Systems from Hewlett-Packard. *Hewlett-Packard.*

[20] Dagna, P. & Hertzer, J. (2013). Evaluation of Multi-threaded OpenFOAM Hybridization for Massively Parallel Architectures. Technical report, Partnership for Advanced Computing in Europe.

[21] Davis, T. A. & Hu, Y. (2011). The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25. ISSN 0098-3500. doi:10.1145/2049662.2049663.

[22] D'Azevedo, E. F., Fahey, M. R. & Mills, R. T. (2005, May). Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. *5th International Conference on Computational Science (ICCS)*, Atlanta, GA, USA.

[23] Dennard, R., Gaensslen, F., Rideout, V., Bassous, E. & Leblanc, A. (1999, April). Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *Proceedings of the IEEE*, 87(4):668–678.

[24] Eghbal, A., Gerber, A. G. & Aubanel, E. (2017). Acceleration of Unsteady Hydrodynamic Simulations Using the Parareal Algorithm. *Journal of Computational Science*, 19:57 – 76. ISSN 1877-7503. doi:http://dx.doi.org/10.1016/j.jocs.2016.12.006.

[25] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K. & Burger, D. (2013). Power Challenges May End the Multicore Era. *Communications of the ACM*, 56(2):93–102.

[26] Ferziger, J. & Perić, M. (1996). *Computational Methods for Fluid Dynamics*. Springer, 3 edition.

[27] G. Karypis (Acc. 2014, June). METIS: Serial Graph Partitioning And Fill-Reducing Matrix Ordering, v5.1.0. Department of Computer Science and Engineering, University of Minnesota, MN, USA. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[28] G. Moore (1998, January). Cramming More Components Onto Integrated Circuits. Proceedings of the *IEEE*.

[29] Gee, M., Siefert, C., Hu, J., Tuminaro, R. & Sala, M. (2006). ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories.

[30] Georgiadis, N. J., Rizzetta, D. P. & Fureby, C. (2010, August). Large-Eddy Simulation: Current Capabilities, Recommended Practices, and Future Research. *AIAA Journal*, 48(8):1772–1784. ISSN 0001-1452. doi:10.2514/1.J050232.

[31] Ghysels, P., Ashby, T. J., Meerbergen, K. & Vanroose, W. (2013). Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines. *Journal of Scientific Computing*, 35(1):48–71.

[32] Gorobets, A., Trias, F. & Oliva, A. (2013, December). A Parallel MPI+OpenMP+OpenCL Algorithm for Hybrid Supercomputations of Incompressible Flows. *Computers & Fluids*, 88:764–772.

[33] Gorobets, A., Trias, F. X., Borrell, R. & Oliva, A. (2014, July). Direct Numerical Simulation of Turbulent Flows with Parallel Algorithms for Various Computing. *6th European Conference on Computational Fluid Dynamics (ECFD VI)*, Barcelona, Spain.

[34] Gropp, W., Kaushik, D., Keyes, D. & Smith, B. (2000). Analyzing the Parallel Scalability of an Implicit Unstructured Mesh CFD Code. In M. Valero, V. Prasanna & S. Vajapeyam, editors, *High Performance Computing HiPC 2000*, volume 1970 of *Lecture Notes in Computer Science*, pages 395–404. Springer Berlin Heidelberg. ISBN 978-3-540-41429-2.

[35] Hager, W. W. (1984). Condition Estimates. *SIAM Journal on Scientific and Statistical Computing*, 5:311–316.

[36] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B. & Vaz, G. (2014, October). Performance Analysis Of Massively-Parallel Computational Fluid Dynamics. *The 11th International Conference on Hydrodynamics (ICHD)*, Singapore.

[37] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B. & Vaz, G. (2014, September). Potential of Chaotic Iterative Solvers for CFD. *The 17th Numerical Towing Tank Symposium (NuTTS 2014)*, Marstrand, Sweden.

[38] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B. & Vaz, G. (2015, June). Chaotic Linear Equation-System Solvers for Unsteady CFD. *The 6th International Conference on Computational Methods in Marine Engineering (MARINE 2015)*, Rome, Italy.

[39] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B. & Vaz, G. (2017, May). On the Strong Scalability of Maritime CFD. *Journal of Maritime Science and Technology*. Accepted for publication.

[40] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B., Vaz, G. & Klaij, C. (TBC). Towards Exascale CFD: Chaotic Multigrid Methods for the Solution of Poisson Equations. *Journal of Computational Physics*. Submitted, under review.

[41] Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J. & Williams, A. (2003). An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories.

[42] Hewlett-Packard Development Company (2014). Scalability of ANSYS 15.0 Applications and Hardware Selection. Technical report.

[43] Hicken, J. & Zingg, D. (2009, June). Globalization Strategies for Inexact-Newton Solvers. In *19th AIAA Computational Fluid Dynamics*, Fluid Dynamics and Co-located Conferences. American Institute of Aeronautics and Astronautics.

[44] Hoekstra, M. (1999). Numerical Simulation of Ship Stern Flows with a Space-marching Navier-Stokes Method. PhD Thesis.

[45] Hoekstra, M. & Vaz, G. (2009, August). The Partial Cavity on a 2D Foil Revisited. In proceedings of the *7th International Symposium on Cavitation*, Ann Arbor, MI, USA.

[46] Horst, S. (2013, May). Why We Need Exascale And Why We Won't Get There By 2020. *Optical Interconnects Conference*, Santa Fe, New Mexico, USA.

[47] Hu, Y. (2005). Efficient, High-Quality Force-Directed Graph Drawing. *Mathematica Journal*, 10(1):37–71.

[48] Hutchinson, B. R. & Raithby, G. D. (1986). A Multigrid Method Based on the Additive Correction Strategy. *Numerical Heat Transfer*, 9:511–537.

[49] Ierotheou, C., Richards, C. & Cross, M. (1989). Vectorization of the SIMPLE Solution Procedure for CFD Problems. *Applied Mathematical Modelling*, 13(9):524 – 529. ISSN 0307-904X.

[50] Intel (2013). Intel Xeon Phi Product Family Performance. Rev 1.2.

[51] Intel (Acc. 2015, April). A Guide to Auto-Vectorization with Intel C++ Compilers.

[52] Klaij, C. & Vuik, C. (2013). Simple-Type Preconditioners for Cell-centered, Collocated, Finite Volume Discretization of Incompressible Reynolds-averaged Navier-Stokes Equations. *International Journal for Numerical Methods in Fluids*, 71(7):830–849.

[53] Klaij, C. M. & Vuik, C. (2012). SIMPLE-type Preconditioners for Cell-centered, Colocated Finite Volume Discretization of Incompressible Reynolds-averaged Navier Stokes Equations. *International Journal for Numerical Methods in Fluids*, 17:830–849. doi:10.1002/fld.

[54] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, S. & Yelick, K. (2008, September). ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. *DARPA IPTO.*

[55] Kuzmin, D. (2010). *A Guide to Numerical Methods for Transport Equations.* University Erlangen-Nürnberg.

[56] Lange, M., Gorman, G., Weiland, M., Mitchell, L. & Southern, J. (2013). Achieving Efficient Strong Scaling with PETSc using Hybrid MPI/OpenMP Optimisation. In J. M. Kunkel, T. Ludwig & H. W. Meuer, editors, *Supercomputing*, volume 7905, pages 97–108. Springer Berlin Heidelberg.

[57] Lee, S., Kim, H., Kim, W. & Van, S. (2003). Wind Tunnel Tests on Flow Characteristics of the KRISO 3,600 TEU Container Ship and 300K VLCC Double-Deck Ship Models. *Journal of Ship Research*, 47(1):24–38.

[58] Lehoucq, R. B., Sorensen, D. C. & Yang, C. (1998). *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems by Implicitely Restarted Arnoldi Methods.* SIAM, Philadelphia, PA, USA.

[59] Liu, X., Smelyanskiy, M., Chow, E. & Dubey, P. (2013). Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS 2013, pages 273–282. ACM, New York, NY, USA. ISBN 978-1-4503-2130-3. doi:10.1145/2464996.2465013.

[60] Löhner, R., Sharov, D., Luo, H. & Ramamurti, R. (2001, January). Overlapping Unstructured Grids. *39th Aerospace Sciences Meeting and Exhibit*, Reno, NV, U.S.A.

[61] Maheswaran, M., Webb, K. & Siegel, H. (1998, August). Reducing the Synchronization Overhead in Parallel Nonsymmetric Krylov Algorithms on MIMD Machines. In proceedings of *International Conference on Parallel Processing*, Minneapolis, Minnesota, USA.

[62] Menter, F., Egorov, Y. & Rusch, D. (2012, February). Steady and Unsteady Flow Modelling Using the KSKL Model. In *Turbulence, Heat and Mass Transfer 5*. Dubrovnik, Croatia.

[63] Menter, F., Kuntz, M. & Langtry, R. (2003, October). Ten Years of Industrial Experience with the SST Turbulence Model. In *Turbulence, Heat and Mass Transfer 4*. Antalya, Turkey.

[64] Mittal, R. & Iaccarino, G. (2005). Immersed Boundary Methods. *Annual Review of Fluid Mechanics*, 37(1):239–261.

[65] Molland, A., Turnock, S. R. & Hudson, D. (2011, September). *Ship Resistance and Propulsion: Practical Estimation of Ship Propulsive Power*. Cambridge University Press.

[66] MPI Standard, Version 3.0 (Acc. 2014, June). `http://www.mpi-forum.org/docs/docs.html`.

[67] Murthy, J. Y. & Mathur, S. (2002). *Numerical Methods in Heat, Mass, and Momentum Transfer*. Purdue University.

[68] Napov, A. & Notay, Y. (2011). Algebraic Analysis of Aggregation-Based Multigrid. *Numerical Linear Algebra with Applications*, 18(3):539–564. ISSN 1099-1506.

[69] Nguyen, N., Peraire, J. & Cockburn, B. (2011). An Implicit High-Order Hybridizable Discontinuous Galerkin Method for the Incompressible Navier-Stokes Equations. *Journal of Computational Physics*, 230(4):1147 – 1170.

[70] Notay, Y. (2008). An Aggregation-Based Algebraic Multigrid Method. *Electronic Transactions on Numerical Analysis*, 37(6):123–146.

[71] NVIDIA Corporation (2010). NVIDIA CUDA C Programming Guide. Version 3.2.

[72] Olson, L. N. & Schroder, J. B. (2011). Smoothed Aggregation Multigrid Solvers For High-order Discontinuous Galerkin Methods For Elliptic Problems. *Journal of Computational Physics*, 230(18):6959 – 6976. ISSN 0021-9991.

[73] OpenACC.org (2015, October). OpenACC Application Programming Interface Version 2.5.

[74] OpenFOAM (2014). OpenFOAM User Guide.

[75] OpenMP Architecture Review Board (2015, November). OpenMP Application Programming Interface Version 4.5.

[76] Pereira, F., Eça, L. & Vaz, G. (2013, June). On the Order of Grid Convergence of the Hybrid Convection Scheme for RANS Codes. In proceedings of *CMNI*, Bilbao, Spain.

[77] Pringle, G. (2010). Porting OpenFOAM to HECToR. EPCC, The University of Edinburgh.

[78] Rabenseifner, R. (2014). Hybrid Parallel Programming with MPI & OpenMP MPI + OpenMP and Other Models. Technical report, High Performance Computing Centre, Stuttgart.

[79] Rijpkema, D. R. (2008, November). Numerical Simulation of Single-Phase and Multi-Phase Flow over a NACA 0015 Hydrofoil. M.Sc. Thesis, Delft University of Technology, Faculty of 3ME.

[80] Rosetti, G., Vaz, G. & Fujarra, A. (2012, July). URANS Calculations for Smooth Circular Cylinder Flow in a Wide Range of Reynolds Numbers: Solution Verification and Validation. *Journal of Fluids Engineering, ASME*, page 549.

[81] Rosetti, G. F., Vaz, G., Hawkes, J. & Fujarra, A. (TBC). Modeling Transition using Turbulence and Transition Models for a Flat Plate Flow: Uncertainty, Verification and Sensitivity. *ASME Journal of Verification & Validation*. Submitted, unpublished.

[82] Rossi, R., Mossaiby, F. & Idelsohn, S. (2013, July). A Portable OpenCL-Based Unstructured Edge-Based Finite Element Navier-Stokes Solver on Graphics Hardware. *Computers & Fluids*, 81:134–144.

[83] Saad, Y. & van der Vorst, H. a. (2000, November). Iterative Solution of Linear Systems in the 20th Century. *Journal of Computational and Applied Mathematics*, 123(1-2):1–33. ISSN 03770427. doi:10.1016/S0377-0427(00)00412-X.

[84] Saltzer, J. & Kaashoek, M. (2009). *Principles of Computer System Design: An Introduction*, volume 1. Massachusetts Institute of Technology, Cambridge, MA, USA.

[85] Shalf, J. (2013, October). The Evolution of Programming Models in Response to Energy Efficiency Constraints. *Oklahoma Supercomputing Symposium*, Norman, Oklahoma, USA.

[86] shwg.org (Acc. 2017, April). Submarine Hydrodynamics Working Group. `http://www.shwg.org`.

[87] Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E. & Mavriplis, D. (2014). CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences. Technical Report March, NASA Langley Research Center, Hampton, VA.

[88] Soukov, S. A., Gorobets, A. V. & Bogdanov, P. (2013, January). OpenCL Implementation of Basic Operations for a High-order Finite-volume Polynomial Scheme on Unstructured Hybrid Meshes. *Procedia Engineering*, 61:76–80. ISSN 18777058. doi:10.1016/j.proeng.2013.07.096.

[89] Spalart, P. & Allmaras, S. (1992). A One-Equation Turbulence Model for Aerodynamics Flows. *AIAA*, (0439).

[90] Spiegel, A., May, D. & Bischof, C. (2006). Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing. In J. Dongarra, K. Madsen & J. Vasniewski, editors, *Applied Parallel Computing: State of the Art in Scientific Computing*, pages 433–440. Springer Berlin Heidelberg.

[91] Steyer, M., Duhem, L. & Fernandez, M. (2015, March). Intel HPC Software Workshop Series, Oxford, UK. [Including personal conversation with Michael Steyer].

[92] Stone, J. E., Gohara, D. & Shi, G. (2010, May). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73. ISSN 0740-7475. doi:10.1109/MCSE.2010.69.

[93] SWIG (Acc. 2017, February). `http://www.swig.org`.

[94] The Platform (Acc. 2015, May). More Knights Landing Xeon Phi Secrets Unveiled. `http://www.theplatform.net/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/`.

[95] The Platform (Acc. 2015, May). Where Will Future Xeon Phi Chips Land? `http://www.theplatform.net/2015/03/26/where-will-future-xeon-phi-chips-land/`.

[96] Top 500 List (Acc. 2017, February). `http://www.top500.org`.

[97] Toxpeus, S., Kuin, R., Kerkvliet, M., Hoeijmakers, H. & Nienhuis, B. (2014, June). Improvement of Resistance and Wake Field of an Underwater Vehicle by Optimising the Fin-Body Junction Flow With CFD. *ASME 33rd International*

Conference on Ocean, Offshore and Arctic Engineering, San Francisco, California, USA.

[98] Turnock, S. (1993). Prediction of Ship Rudder-Propeller Interaction Using Parallel Computations and Wind Tunnel Measurements. PhD Thesis.

[99] Vaz, G., Jaouen, F. & Hoekstra, M. (2009, May–June). Free-Surface Viscous Flow Computations: Validation of URANS Code FRESCO. *28th International Conference on Ocean, Offshore and Arctic Engineering (OMAE)*, Honolulu, Hawaii, USA.

[100] Versteeg, H. K. & Malalasekera, W. (1995). *An Introduction to Computational Fluid Dynamics The Finite Volume Method*. Pearson Education, 2 edition. ISBN 9788131720486.

[101] VI-HPS, Score-P, v.1.2.3 (Acc. 2013, November). `http://www.vi-hps.org/projects/score-p`.

[102] Wang, Z. J. & Parthasarathy, V. (2000). A Fully Automated Chimera Methodology for Multiple Moving Body Problems. *International Journal for Numerical Methods in Fluids*, 33(7):919–938. ISSN 1097-0363.

[103] Wesseling, P. (1992). *An Introduction to Multigrid Methods*. Pure and Applied Mathematics. John Wiley & Sons Australia, Limited. ISBN 9780471930839.

[104] Weymouth, G. & Yue, D. K. (2011). Boundary Data Immersion Method for Cartesian-Grid Simulations of Fluid-Body Interaction Problems. *Journal of Computational Physics*, 230(16):6233 – 6247.

[105] Windt, J. (2013, September). Adaptive Mesh Refinement in Viscous Flow Solvers: Refinement in the Near-wall Region, Implementation and Verification. *16th Numerical Towing Tank Symposium (NuTTS), Mllheim, Germany.*

[106] Yang, X. & Mittal, R. (2014). Acceleration of the Jacobi Iterative Method by Factors Exceeding 100 Using Scheduled Relaxation. *Journal of Computational Physics*, 274(0):695–708. doi:http://dx.doi.org/10.1016/j.jcp.2014.06.010.

[107] Zhiyin, Y. (2015). Large-eddy simulation: Past, Present and the Future. *Chinese Journal of Aeronautics*, 28(1):11–24. ISSN 1000-9361. doi:10.1016/j.cja.2014.12.007.

[108] Zuo, X.-Y., Zhang, L.-T. & Gu, T.-X. (2014, December). An Improved Generalized Conjugate Residual Squared Algorithm Suitable for Distributed Parallel Computing. *Journal of Computational and Applied Mathematics*, 271:285–294.

Appendices

Proceedings of the 11th International Conference on Hydrodynamics (ICHD 2014)
October 19 – 24, 2014
Singapore

PERFORMANCE ANALYSIS OF MASSIVELY-PARALLEL COMPUTATIONAL FLUID DYNAMICS

J. HAWKES

Fluid Structure Interactions (FSI), University of Southampton, UK &
MARIN Academy, Maritime Research Institute Netherlands, Netherlands

S.R.TURNOCK[†], S.J. COX[*], A.B. PHILLIPS[†]

Fluid Structure Interactions (FSI) [†] and Computational Engineering & Design (CED)[*],
University of Southampton, UK

G. VAZ

R&D Department, Maritime Research Institute Netherlands, Netherlands

As modern supercomputers edge towards exascale, their architectures are becoming more parallel. In order for computational fluid dynamics (CFD) simulations to operate efficiently on newer machines, a complete harmony between hardware, software and numerical algorithms is required. In the work presented here, a typical CFD code is instrumented, and a strong-scalability study performed to identify areas of the execution which require improvement, using the well-known KVLCC2 test case. The effects of changing discretization schemes, mesh structure, turbulence models and linear solvers are all tested. The results show that data-exchange among cores and the inner-loop pre-conditioners both have a large impact on performance in a massively-parallel environment, and should be the focus of future developments.

1. Introduction

The history of the "Top500" supercomputers [1] in figure 1 shows the exponential growth of computing power available to CFD since 1993. The maximum performance, based on a linear algebra tool (LINPACK), has doubled approximately every 14 months, allowing maritime simulations of up to 32-billion cells to be performed [2]. The way in which this growth is achieved varies over time, and programming paradigms for CFD must change as necessary to ensure efficient scaling.

Pre-2004, floating-point operation (FLOP) rates grew exponentially as transistor size decreased. In 2004, predictions on the power consumption of central-processing units (CPUs or simply “chips”) broke down, and since this point the growth of computational performance has been limited by power requirements [3]. CPUs with chip-level multiprocessing (CMP) have become the norm, packaging multiple cores on one chip to improve efficiency. Figure 2 shows this shift in growth mechanism.

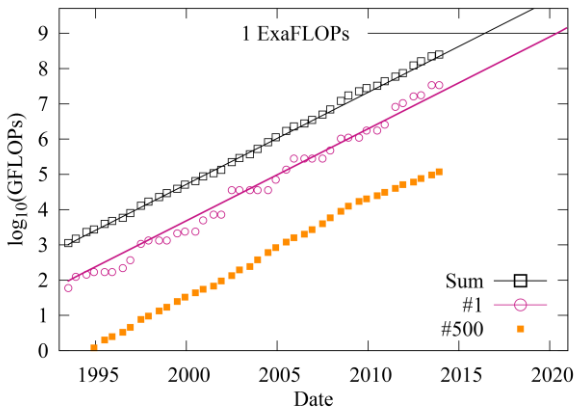


Figure 1: Performance of the Top500 [1] supercomputers. Showing the #1, #500 and sum LINPACK max performance over 20 years, with exponential trend lines.

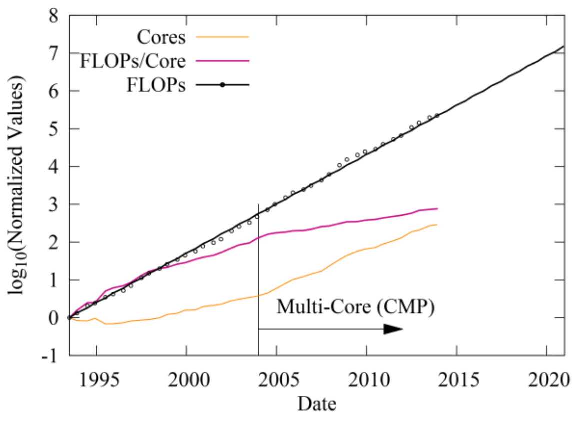


Figure 2: Changes in FLOPs/core, total FLOPs and number of cores, normalized to 1993 values. Values are an average of the Top500 data [1].

CFD has benefited from good "weak scalability" for the last 20 years. As the number of cores has increased CFD problem-sizes have increased too, such that the efficient ratio of cells-per-core has remained roughly constant (anywhere from 20k-100k cells per core, depending on the code, supercomputer and the problem complexity). By current trends, parallelization will be around 14-times greater in 6 years, and one could expect CFD simulations to also be 14-times larger, but there are great challenges in reaching this level.

In 2008, a group of over 20 experts in the supercomputing field were commissioned by DARPA (Defense Advanced Research Projects Agency) to assess the challenges associated with creating the first exascale machine (capable of 1 ExaFLOPs) [4] which should be viable in 2020 according to current growth. Their predictions, which have remained valid over the last 6 years, explain that as power efficiency continues to improve (in terms of pico-Joules per FLOP) the power required to run inter-nodal communication becomes dominant.

Since 2011, there has been a large push towards many-core nodes, which feature O(100) low-power cores to increase efficiency and total FLOP rates using many-core co-processor or graphics-card accelerators [3]. The inter-nodal power bottleneck is likely to drive this trend further and increase intra-node concurrency to O(1k) or O(10k) cores per node [4,5]. With decreasing core clock rates of these many-core nodes, total concurrency on an exascale machine is likely to be 300-times that of the current #1 Top500 machine - Tianhe 2.

Comparing this growth in cores (300x) to the expected growth in CFD complexity (14x), it is clear that there is a need to improve the "strong scalability" of CFD codes. By 2020, it will be necessary to reduce the efficient cells-per-core ratio to approximately 5% of its current value, if predictions hold true.

This paper aims to break down the scalability of a CFD code into its fundamental parts – using the well-known KVLCC2 test case. Profiling tools will be used to extract information on the run-time of the code whilst various user-settings are changed, in an attempt to demystify their effects on scalability. Assessments will be made to determine the areas of the program which exhibit poor strong scalability, as part of ongoing research which aims to develop new techniques to improve the minimum cells-per-core ratio.

2. Experimental Setup

In this section, an overview of the CFD code (ReFRESCO), the supercomputer (IRIDIS4) and the profiling tool (Score-P) is given. The test-case setup is also detailed, and the key scalability studies performed in this paper are explained.

2.1. *ReFRESCO*

ReFRESCO is a viscous-flow CFD code that solves multiphase (unsteady) incompressible flows with the Reynolds-averaged Navier-Stokes (RANS) equations, complemented with turbulence models, cavitation models and volume-fraction transport equations for different phases [6]. The equations are discretized in strong-conservation form using a finite-volume approach with cell-centered collocated variables. The SIMPLE algorithm is used to ensure mass conservation, with pressure-weighted-interpolation (PWI) to tackle the pressure-velocity decoupling issue arising from the collocated arrangement [7].

Time integration is performed implicitly with first or second-order backward schemes. At each implicit time step, the non-linear system for velocity and pressure is linearized with Picard's method - and a segregated or coupled method applied. A segregated approach is adopted for the solution of all other transport equations. All non-linearity is tackled by means of an iterative process so-called the *outer loop*. For each outer-loop iteration, and for each transport equation, an algebraic system of linear equations is solved iteratively until a prescribed residual decay is achieved - this iterative process is the *inner loop*. In order to increase the robustness of the iterative process, implicit (directly on the left-hand-side matrix diagonal) and explicit (directly on the new solution for each variable) relaxation schemes are used.

All numerical schemes used to discretize the transport equations (convection schemes, diffusion, gradients, non-orthogonality corrections, eccentricity corrections) apply their low-order contributions implicitly, to the left-hand side of the equation system; and their higher-order contributions explicitly, to the right-hand side of the system, using values from the previous outer loop.

The implementation is face-based, which permits grids with elements consisting of an arbitrary number of faces and hanging nodes. This also means that all grids, being them structured or unstructured, are treated in the same way inside ReFRESCO, even if usually for unstructured grids more outer-loop iterations are needed to take into account the lower quality metric characteristics.

The code is parallelized using MPI (Message Passing Interface) and sub-domain decomposition. The grids are partitioned in sub-domains, each one having a layer of common cells so-called *ghost-cells*. Each of these sub-domains is calculated in its own MPI process. The ghost-cells are treated as normal cells, as far as the numerical algorithms are concerned, and are therefore handled implicitly.

In many ways, ReFRESCO represents a general-purpose CFD commercial code, with state-of-the-art features such as moving, sliding and deforming grids and automatic grid refinement - but it has been verified, validated and optimized for numerous maritime industry problems.

ReFRESCO is currently being developed at MARIN (Netherlands) [8] in collaboration with IST (Portugal) [9], USP-TPN (University of Sao Paulo, Brazil) [10], TUDelft (Technical University of Delft, the Netherlands) [7], RuG (University of Groningen, the Netherlands) [11] and recently at UoS (University of Southampton, UK).

2.2. *IRIDIS4*

ReFRESCO will be run on the University of Southampton's latest supercomputer, IRIDIS4, which was ranked #179 on the Top500 list of November 2013 [1]. IRIDIS4 has 750 compute nodes, consisting of two Intel Xeon E5-2670 Sandybridge processors (8 cores, 2.6 Ghz), for a total of 12,200 cores and a maximum performance of 227 TFLOPs. Each node is diskless, but is connected to a parallel file system, and has 64GB of memory. The nodes run Red Hat Enterprise Linux (RHEL) version 6.3. Nodes are grouped into sets of ~30, which communicate via 14 Gbit/s Infiniband. Each of these groups is connected to a leaf switch, and inter-switch communication is then via four 10 Gbit/s Infiniband connections to each of the core switches. Management functions are controlled with a GigE network.

2.3. *Profiling Tools*

ReFRESCO will be instrumented with Score-P [12], a profiling tool developed by VI-HPS (Virtual Institute – High Performance Supercomputing). The tool provides compile-time wrappers for ReFRESCO and run-time configuration options which provide useful information on program execution. The results show total call-counts and total time spent within certain functions, and also give insight into MPI communications and load-balancing.

All forms of executable instrumentation are intrusive, since the measuring and buffering of various timers introduces some overhead. Care must be taken to filter the profiling tool, such that only useful information is produced and the overhead is minimized. Profiling tools also disable some compiler optimization (most notably function in-lining), which may have some repercussions.

The total run-time of a profiled ReFRESCO compared to a non-profiled ReFRESCO differs by approximately 2%, and it is assumed that this does not affect the results of the scalability studies.

2.4. *Test Case*

The test case for this investigation will be the KVLCC2 double-body wind tunnel model [13]. The model is simulated at a Reynolds number of $4.6\text{x}10^6$. The domain and boundary conditions are shown in figures 3 and 4.

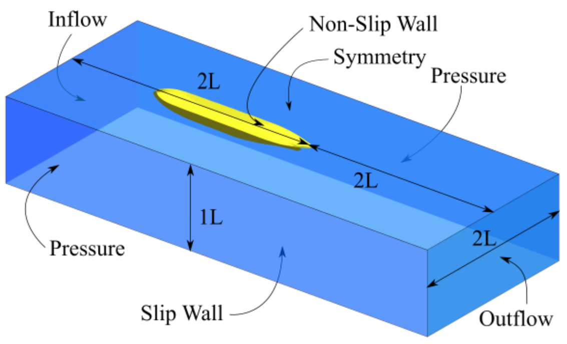


Figure 3: The KVLCC2 domain, not to scale.

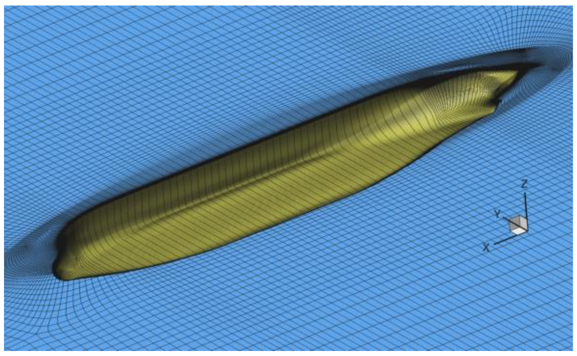


Figure 4: An example of the structured mesh (2.67m cells), from below the waterline.

The following parameters have been used as a baseline for scalability studies, although some variables are tested independently in following studies:

- All tests use a segregated solver.
- Unless specified, a grid of 2.67m cells is used, which covers the current efficient cells-per-core ratio range (20k on ~133 cores, 100k on ~27 cores) with high resolution, and also shows the trends when moving to lower cells-per-core ratio.
- A k-ω, two-equation shear-stress transport turbulence model (SST-2003) is used [14].
- The x-, y- and z-momentum equations, and the turbulence equations, are solved with a Block Jacobi pre-conditioner and a generalized minimal residual (GMRES) solver from the PETSc suite [15]. The inner loop relative ℓ^2-norm convergence tolerance (henceforth ILCT) is provisionally set to 1% (0.01). An explicit outer-

loop relaxation factor of 0.15 is used, as well as an implicit relaxation factor which begins at 0.8 and ramps up to 0.85 over the first 100 iterations.

- The pressure equation is solved with a multi-grid pre-conditioner (ML) and a GMRES solver, using an explicit relaxation factor of 0.1. The ILCT is also chosen as 1%.
- The default convection discretization scheme for the momentum equations is QUICK (Quadratic Upstream Interpolation for Convective Kinematics) with a flux limiter [16], and first-order upwind for the turbulence equations. For all equations, the diffusion terms are discretized using a 2nd-order central scheme. The gradients of any variable (velocity, pressure, turbulence quantities) are discretized using a 2nd-order Gauss-theorem scheme.

2.5. *Scalability Studies*

Initially, a scalability study will be performed using the default settings to provide a benchmark and basic understanding. Following this, various parameters will be changed which represent common user settings, in an attempt to determine their effect, or lack thereof, on strong scalability. The effect of momentum convective flux discretization scheme will be observed, as well as the effects of using different turbulence models, different grids (structured vs. unstructured) and finally, different pre-conditioners and solvers.

3. Scalability Study: A Breakdown of ReFRESCO

In this section, a basic scalability study inheriting all the default settings from section 2.5 is performed over 1000 iterations. Profiling tools are used to break down the execution into distinct sets of subroutines:

- *Assemble* – the assembly of each inner-loop equation system, including construction of the right-hand side vector, for each transport equation.
- *Solve* – the pre-conditioner and solver used to iterate the systems of linear equations.
- *Gradients* – the calculation of gradient values following the iterative solution to each transport equation.
- *Exchange* – data exchange between ghost cells, including transported values, gradients, and other computed values such as eddy viscosity.
- *Other* – the remainder of the above, including initialization routines.

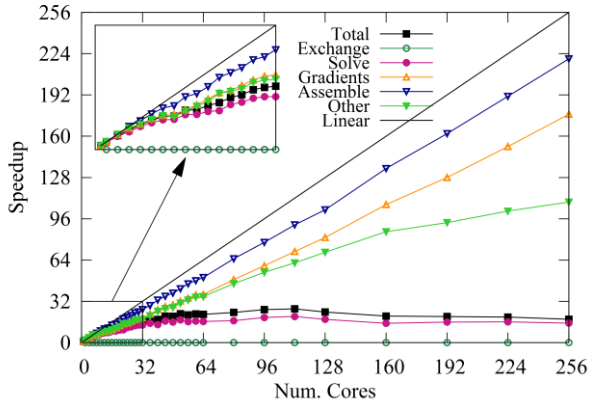


Figure 5: Scalability breakdown of a typical CFD simulation

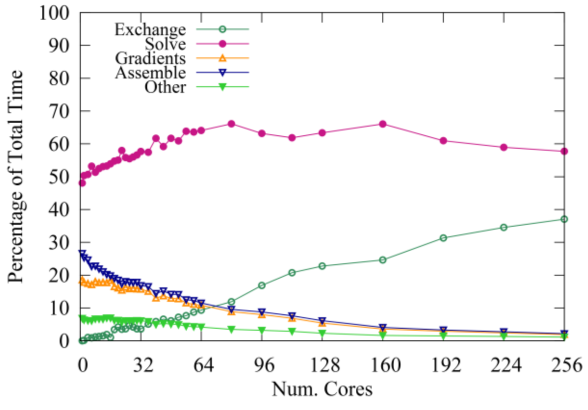


Figure 6: Proportion of total wall-time spent in different subroutines.

Figure 5 shows a typical scalability plot, where the "speedup" is defined as the relative decrease in wall-time with N cores compared to the serial runtime. This speedup can be compared to a linear ("ideal") speedup. Figure 6 shows the proportions of the total wall-time occupied by different sets of subroutines as the number of cores increases.

Figure 5 shows that the *assemble*, *gradients* and *other* parts of the code scale reasonably well, with *other* dropping off at higher core counts due to a number of parallel broadcasts and higher MPI initialization cost. At higher core-counts these parts of the code become a very small proportion of the total run-time due to their efficient scaling, as shown in figure 6. The *exchange* portion of the execution scales inversely with number of cores, which causes a serious bottleneck on strong-scalability. The *solve* routines occupy 48% of the total wall-time in serial operation, and also scale poorly. These are both areas requiring improvement.

4. Scalability Study: Convective Discretization Scheme

The previous scalability study was performed with a QUICK momentum convective discretization scheme with a flux limiter. In the following study, this discretization scheme was varied to ensure that the scalability was independent of the chosen scheme. A first order upwind (UD1), a 50%-central-50%-upwind blend (CD5-UD5), a 90%-central-10%-upwind blend (CD9-UD1), QUICK with limiter and QUICK without limiter (QUICK-NL) were used on a range of structured grids from 317k to 23.4m cells. The solution was allowed to reach an outer loop ℓ^{infty}-norm convergence of 10^{-5}.

The computed form factor [17] is compared to wind-tunnel experimental results [13] in figure 7. The results are presented against the grid coarsening factor, which is the ratio of cells compared to the finest grid.

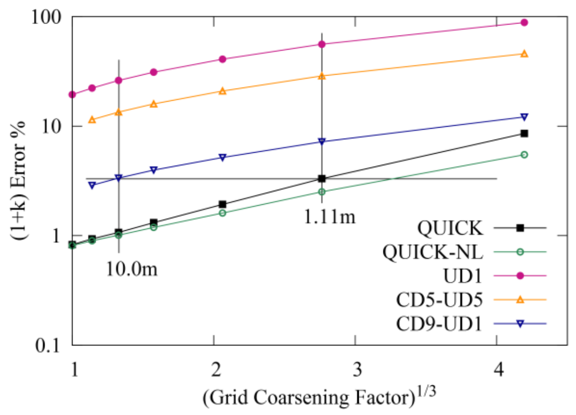


Figure 7: Grid convergence study using various momentum convection discretization schemes, showing form-factor error percentage when compared to experimental results [19].

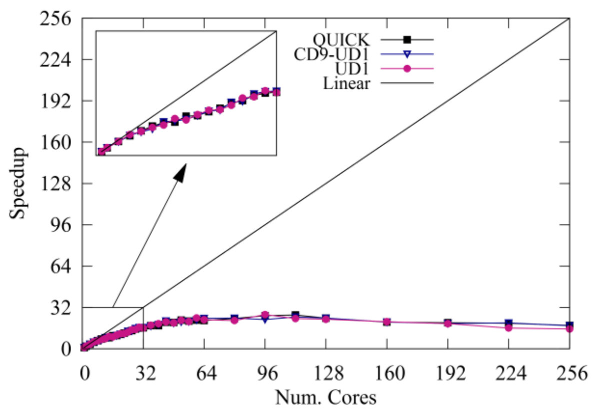


Figure 8: Total speedup for various momentum convection discretization schemes.

As expected, the higher-order schemes provide much higher levels of accuracy, with QUICK matching CD9-UD1 using far fewer cells (1.11m vs. 10.0m). The amount of wall-time required to reach convergence grew exponentially with number of cells, as the number of iterations and the time-per-iteration both increased, but on similar grids all the schemes took similar computational time.

More importantly, the discretization schemes were also compared in a scalability study. The simulations ran for 1000 iterations on a grid of 2.67m cells. The results are shown in figure 8 – no significant changes in scalability were noticed, suggesting that there will be no issues with higher-order schemes in a massively-parallel environment.

Note that ReFRESCO implements a 2nd-order accurate QUICK scheme, since internally the mesh is treated as unstructured – and neighbors-of-neighbors information is not readily available.

5. Scalability Study: Turbulence Models

In this study, the default turbulence model (k-ω SST 2003 [14]) will be compared to a square-root-*kL* (SKL) model [18] and a Spalart-Allmaras (SA) model [19]. Differences are expected due to different amounts of data being computed and exchanged. The simulations are run for 1000 iterations and the speed-up is shown in figure 9.

The SA model required 71% of the total serial run-time of the two-equation models (which were virtually identical in run-time). The SA model has only one transport equation to solve, but also does not incur the overhead of eddy-viscosity and strain-rate computations which, combined, accounts for this wall-time reduction.

Normalized to the serial run-time of each turbulence model respectively, the SA model yields much better scalability. The removal of the second transport equation, eddy viscosity and strain rate computations reduces the amount of data exchange required per outer loop: 2 exchanges versus 8 for the two-equation models (the remaining momentum and pressure equations require a total of 8 exchanges).

6. Scalability Study: Structured vs. Unstructured Grids

Structured grids are often not feasible for practical applications, where the cost of creating the grid is too high. In this study, the scalability of a structured grid simulation is compared to an unstructured one. Unfortunately, it was not possible to create two grids of exactly the same number of cells. An unstructured grid of 12.5m cells was used, and the scalability of two structured grids (10.0m and 15.8m) was linearly interpolated to obtain comparisons. This method is not the most accurate, but any obvious changes in scalability should be clearly visible.

Once again, the simulation was performed to 1000 iterations, but with a more relaxed outer-loop of 0.25 explicit and 0.7 implicit. The results are shown in figure 10.

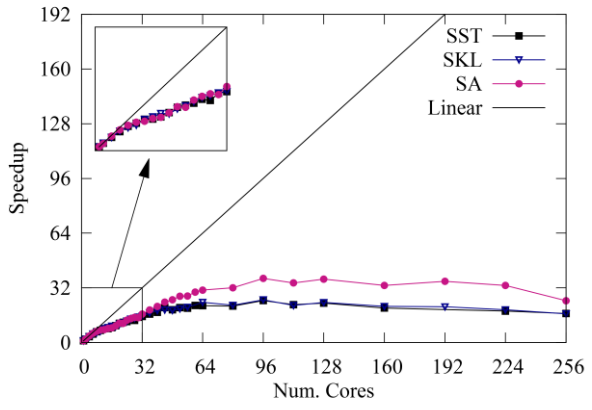


Figure 9: Total speedup for various turbulence models.

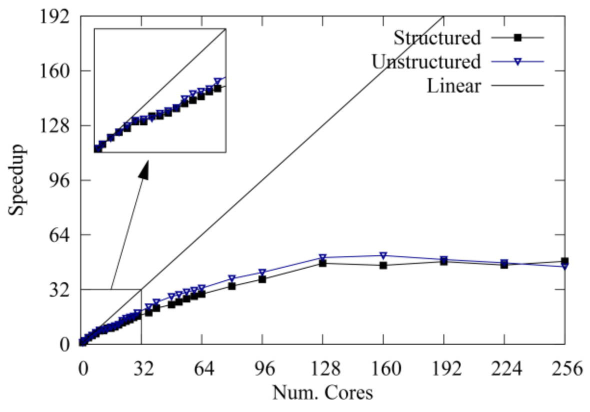


Figure 10: Total speedup with different grids structures.

It is clear that the difference between an unstructured and structured grid is small in terms of scalability. In terms of absolute wall-time, the differences between the two are all less than 5%. This is expected, since ReFRESCO treats the two identically, and the minor differences (in scalability and absolute time) are probably due to the errors associated with linearly interpolating the results.

7. Scalability Study: Pre-conditioners and Solvers

In section 3, the *solve* subroutines were shown to scale poorly, but there are many combinations of pre-conditioner and solver which may perform better. In this scalability study, a number of these combinations will be tested.

The *solve* routines are responsible for iterating the set of linear equations created for each transport equation. The equations are iterated until the solution vector reaches an ℓ^2-norm residual that is a proportion of its initial value. This proportion is the inner loop convergence tolerance (ILCT) – and changing this value greatly affects the load on the solver. It was important to find a reasonable and efficient value for the ILCT before the pre-conditioners and solvers could be tested fairly.

In these tests, starting from a restarted solution of 100 iterations with the default settings, the ILCT is varied between 0.5 (50%) and 10^{-6} (0.0001%). Two outer loop relaxation schemes are tested: implicit 0.9 with explicit 0.1 (tight), and implicit 0.8 with explicit 0.2 (relaxed). In this way, the computational effort associated with linearity and non-linearity of the transport equations can be changed by shifting the focus to the inner loop or outer loop respectively. The total time to reach 10^{-5} ℓ^{infty}-norm outer loop convergence was observed.

Figure 11 shows the results of this study. In general, reducing the inner loop convergence tolerance provides no benefit to overall convergence and serves only to increase the time-per-iteration – except in the case where the linear set of equations is not solved well enough to allow the non-linear iterations to converge. It seems that an ILCT of 0.01 (1%) is a suitable value for numerical stability and efficiency for the outer loop relaxation chosen in section 2.4.

Using this result, the pre-conditioners and solvers can be tested, employing methods from the PETSc suite [15]. For momentum and turbulence, a Block Jacobi, a parallel multi-grid method (BoomerAMG), a parallel additive Schwarz method (ASM), and a parallel ILU pre-conditioner (EUCLID) are tested. For the elliptic pressure equation the same methods are used, but the multi-grid method is replaced with the ML pre-conditioner, which should be faster for elliptic equations.

Two solvers are tested with these combinations: a generalized minimal residual method (GMRES) and a stabilized bi-conjugate gradient squared method (BiCGSTAB).

The scalability results are shown in figure 12, where the key indicates the momentum-turbulence pre-conditioner, the pressure pre-conditioner and the solver for both, respectively. With all pre-conditioners, GMRES and BiCGSTAB perform similarly, so the BCGS results have been removed from figure 12 for clarity. Only the speed-up of the *solve* subroutines are plotted in this graph, having been extracted with the profiling tools. The speed-ups are calculated relative to the serial run-time of BJAC-BJAC-GMRES, in order to make quantitative assessments between total wall-times – the large differences between the pre-conditioners is clearly visible.

Even on low numbers of cores, the Block Jacobi pre-conditioner outperformed all other pre-conditioners, and manages to maintain good speedup to 256 cores. Block Jacobi is a naïve pre-conditioner compared to the others, but the simplicity works in its favour, leading to an overall speedup. EUCLID and BoomerAMG both offered poor scalability, while the additive Schwarz and ML methods offered mediocre scalability. Some of the poorer methods could not complete on a low number of cores, as they exceeded wall-time limitations.

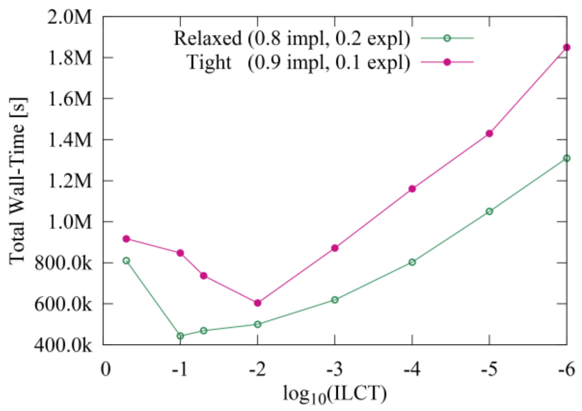


Figure 11: Effects of ILCT on total wall-time using two different outer loop relaxation factors.

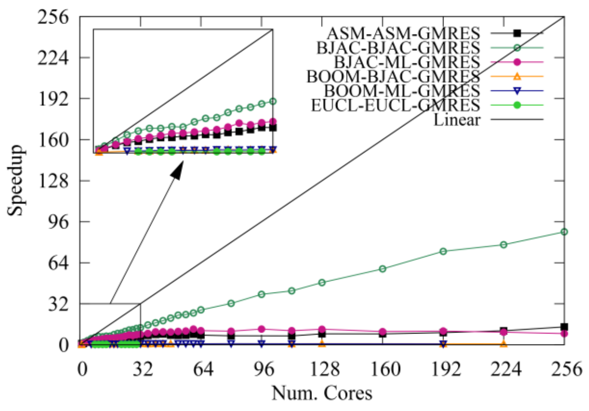


Figure 12: *Solve* scalability with different pre-conditioners and solvers, normalized to BJAC-BJAC-GMRES.

From these studies, it is clear that a Block Jacobi pre-conditioner for all equations is the best starting point for highly scalable CFD. The results of the initial breakdown of ReFRESCO (section 3) are repeated here using these pre-conditioners and the GMRES solver. Figure 13 shows that the *solve* routines have become less of a bottleneck; but in figure 14 it is clear that these routines still take a large part of the execution time. Ignoring the *exchange* routines, which must be improved, *solve* occupies 56% of the remaining execution time on 256 cores.

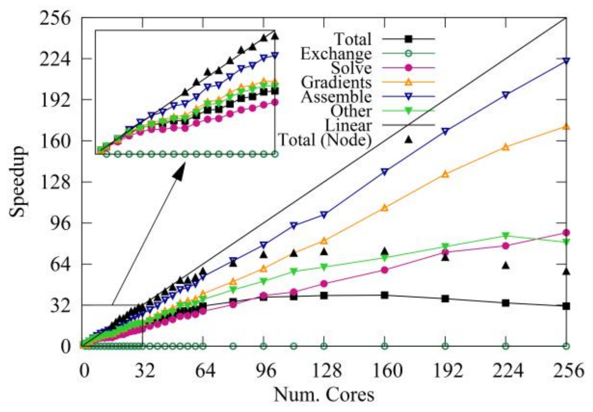


Figure 13: A scalability breakdown of ReFRESCO using more-scalable pre-conditioners (Block Jacobi).

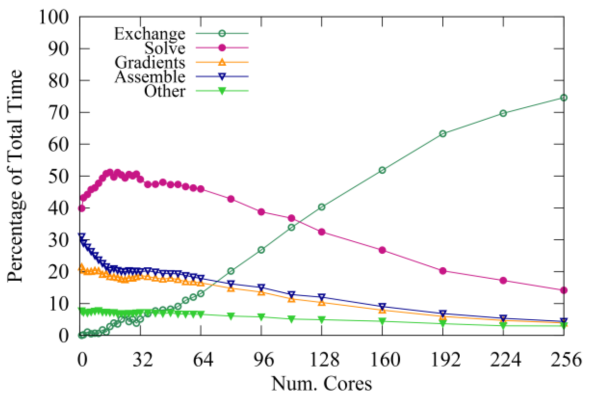


Figure 14: Proportions of total wall-time spent in different routines using more-scalable pre-conditioners (Block Jacobi)

Often, this speed-up is plotted relative to the number of nodes only, such that rather than normalizing to serial run-time, the results are normalized to the run-time on one node (see Figure 13 – “Total (Node)”). This hides the intra-node inefficiency and shows that inter-node scaling is relatively good up to ~42k cells per core (64 cores). For next-generation computing the pressure is on intra-node concurrency, so this view is not the most helpful; however it is important to note that this inter-node scaling compares well with other codes. STAR-CCM+ [20] and ANSYS CFX/FLUENT [21] both show linear speed-up with number of nodes similar to ReFRESCO, but do not present data beyond the efficient range. The ANSYS results also show similar intra-node efficiency (10-12x speedup on 16 cores) compared to ReFRESCO (~10x speedup on 16 cores) – but it is hard to assess fairly without identical hardware and case setup.

8. Conclusions

The breakdown of a typical CFD code was performed in section 3, showing the scalability of the code and the cost of various routines at higher core-counts. It was noted that the *exchange* routines responsible for updating ghost-cell values between cores were costly and non-scalable. There will be a need to explore alternative communication paradigms for massively-parallel CFD – perhaps taking more advantage of shared (intra-node) memory.

The choice of mesh structure and discretization scheme was shown to have very little effect on strong scalability, which is promising for next-generation higher-order CFD simulations. Turbulence models which required more communication caused decreased overall scalability, but improvements in the *exchange* routines may negate this.

Finally, the choice of pre-conditioner had a large effect on the scalability of the code. The more complex pre-conditioners, such as the multi-grid methods, scaled poorly compared to the straight-forward Block Jacobi method. The solver itself seems to have little effect, although only GMRES and BICGStab were tested.

Even the best pre-conditioners caused the *solve* routines to scale poorly compared to the *assembly* and *gradients* routines, and the *solve* routines also take the largest proportion of wall-time. This should be an area for development – especially considering the large differences between the existing pre-conditioners.

Although attempts have been made to generalize the results obtained in these studies, it is important to note that the complex interaction between parts of the code make it difficult to do so. Moreover, only a simple KVLCC2 case has been tested here, and it is assumptive to extrapolate these results to more complex cases – especially where extra features are added, such as cavitation models, free-surfaces or moving interfaces. In addition, some significant parameters have not been tested – such as the use of a coupled solver.

Despite this, there are certain parts of the code such as the *solve* routines and ghost-cell *exchange* routines that can be identified as areas for improvement. These should be tackled in order to allow CFD simulations to continue to benefit from next-generation supercomputing facilities.

Acknowledgments

The authors acknowledge the use of the IRIDIS HPC Facility, and associated support services at the University of Southampton. Approximately 150k core-hours were used for this work over ~700 jobs.

References

1. *Top500*, top500.org (accessed Feb. 2014).
2. T. Nishikawa, Y. Yamade, M. Sakuma, C. Kato, *Fully resolved large eddy simulation as an alternative to towing tank tests – 32 billion cells computation on K computer*, Numerical Towing Tank Symposium (NUTTS), Mülheim, Germany, September 2nd-4th (2013).
3. H. Simon, *Why we need exascale and why we won't get there by 2020,* Optical Interconnects Conference, Santa Fe, New Mexico, August 27th (2013).
4. P.M. Kogge (editor), *Exascale computing study: technology challenges in achieving exascale systems*, Univ. of Notre Dame, CSE Dept. Tech. Report TR-2008-13 (2008).
5. J. Shalf, *The evolution of programming models in response to energy efficiency constraints*, Oklahoma Supercomputing Symposium, Oklahoma, USA, October 2nd (2013).
6. G. Vaz, F. Jaouen and M. Hoekstra, *Free-surface viscous flow computations. Validation of URANS code FRESCO*, In Proceedings of OMAE, Hawaii, USA, May 31st-June 5th (2009).
7. C.M. Klaij and C. Vuik, *Simple-type preconditioners for cell-centered, collocated, finite volume discretization of incompressible Reynolds-averaged Navier-Stokes equations*, International Journal for Numerical Methods in Fluids, 71(7), pp. 830-849 (2013).
8. L. Eca and M. Hoekstra, *Verification and validation for marine applications of CFD*, 29th Symposium on Naval Hydrodynamics (ONR), Gothenburg, Sweden, August 26th-31st (2012).
9. F. Pereira, L. Eca and G. Vaz, *On the order of grid convergence of the hybrid convection scheme for RANS codes*. CMNI, pp. 1–21, Bilbao, Spain, June 25th-28th (2013).
10. G.F. Rosetti, G. Vaz, and A.L.C. Fujarra, *URANS calculations for smooth circular cylinder flow in a wide range of Reynolds numbers: solution verification and validation*, Journal of Fluids Engineering, ASME, July, p. 549 (2012).
11. H. Bandringa, R. Verstappen, F. Wubbs, C. Klaij and A.v.d. Ploeg, *On novel simulation methods for complex flows in maritime applications,* NUTTS, Cortona, Italy, October 7th-9th (2012).
12. VI-HPS, *Score-P,* vi-hps.org/projects/score-p, version 1.2.3, (accessed Nov. 2013).
13. S. Lee, H. Kim, W. Kim and S. Van, *Wind tunnel tests on flow characteristics of the KRISO 3,600 TEU containership and 300K VLCC double-deck ship models,* Journal of Ship Research, 47(1), pp. 24-38 (2003).
14. F.R. Menter, M.Kuntz, and R. Langtry, *Ten years of industrial experience with the SST turbulence model,* Turbulence, Heat and Mass Transfer 4, Antalya, Turkey, October 12th-17th (2003).
15. S. Balay et al., *PETSc user manual,* mcs.anl. gov/petsc, Argonne National Labatory, ANL-95/11 – Rev. 3.4 (2013).
16. M. Hoekstra, *Numerical simulation of ship stern flows with a space-marching Navier-Stokes method,* PhD Thesis, Delft University of Technology (1999).
17. A.F. Molland, S.R. Turnock and D.A. Hudson, *Ship resistance and propulsion: practical estimation of ship propulsive power*, Cambridge, GB, Cambridge University Press, p.70 (2013)
18. F.R. Menter, Y. Egorov and D. Rusch, *Steady and unsteady flow modelling using the* $k-\sqrt{kL}$ *model,* Turbulence, Heat and Mass Transfer 5, Dubrovnik, Croatia, September 25th-29th (2006).
19. P.R. Spalart and S.R. Allmaras, *A one-equation turbulence model for aerodynamic flows*, AIAA Paper 92-0439 (1992).
20. STAR-CCM+, *Performance benchmark and profiling*, HPC Advisory Council (Best Practices), July (2010).
21. D. Mize, *Scalability of ANSYS applications on multi-core and floating point accelerator processor systems from Hewlett-Packard,* Hewlett-Packard, February 7th (2012).

Potential of Chaotic Iterative Solvers for CFD

J. Hawkes^{a,c*}, **G. Vaz**^{b}, **S. R. Turnock**^{c}, **S. J. Cox** ^{d}, **A. B. Phillips**^{c}

^{a}MARIN Academy; ^{b}Research and Development Department; MARIN, Wageningen, NL.
^{c}Fluid Structure Interactions (FSI); ^{d}Computational Engineering and Design (CED);
University of Southampton, UK.

1 Introduction

Computational Fluid Dynamics (CFD) has enjoyed the speed-up available from supercomputer technology advancements for many years. In the coming decade, however, the architecture of supercomputers will change, and CFD codes must adapt to remain current.

Based on the predictions of next-generation supercomputer architectures it is expected that the first computer capable of 10^{18} floating-point-operations-per-second (1 ExaFLOPS) will arrive in around 2020. Its architecture will be governed by electrical power limitations, whereas previously the main limitation was pure hardware speed. This has two significant repercussions [14, 17, 25]. Firstly, due to physical power limitations of modern chips, core clock rates will decrease in favour of increasing concurrency. This trend can already been seen with the growth of accelerated "many-core" systems, which use graphics processing units (GPUs) or co-processors. Secondly, inter-nodal networks, typically using copper-wire or optical interconnect, must be reduced due to their proportionally large power consumption. This places more focus on shared-memory communications, with distributed-memory communication (predominantly MPI - "Message Passing Interface") becoming less important.

The current most powerful computer, Tianhe-2 [26], capable of 33 PFlops, consists of 3,120,000 cores. The first exascale machine, which will be 30 times more powerful, is likely to be 300-times more parallel – which is a massive acceleration in parallelization compared to the last 50 years. This concurrency will come primarily from intra-node parallelization. Whereas Tianhe-2 features an already-large O(100) cores per node, an exascale machine must consist of O(1k-10k) cores per node.

CFD has benefited from *weak scalability* (the ability to retain performance with a constant elements-per-core ratio) for many years; its *strong scalability* (the ability to reduce the elements-per-core ratio) has been poor and mostly irrelevant. With the shift to massive parallelism in the next few years, the strong scalability of CFD codes must be investigated and improved.

In this paper, a brief summary of earlier results [12] is given, which identified the linear-equation system solver as one of the least-scalable parts of the code. Based on these results, a chaotic iterative solver, which is a totally-asynchronous, non-stationary, linear solver for high-scalability, is proposed. This paper focuses on the suitability of such a solver, by investigating the linear equation systems produced by typical CFD problems. If the results are optimistic, future work will be carried out to implement and test chaotic iterative solvers.

2 ReFRESCO

The work presented in this paper focuses on the development of ReFRESCO – a typical viscous-flow CFD code. ReFRESCO solves multiphase, unsteady, incompressible flows with the Reynolds-Averaged Navier Stokes (RANS) equations, complemented with turbulence models, cavitation models and volume-fraction transport equations for different fluid phases [27]. ReFRESCO represents a general-purpose CFD code, with state-of-the-art features such as moving, sliding and deforming grids and automatic grid refinement – but has been verified, validated and optimized for numerous maritime industry problems.

The RANS equations are discretized in strong conservation form using a finite-volume approach with cell-centred collocated variables. The SIMPLE algorithm is used to ensure mass conservation, with pressure-weighted interpolation (PWI) to tackle pressure-velocity decoupling issue arising from the collocated arrangement [16].

Time integration is performed implicitly with first or second-order backward schemes. At each time step, the non-linear system for velocity and pressure is linearized with Picard's method – and a segregated method applied. All non-linearity is tackled by means of an iterative process so-called the outer loop. For each outer-loop iteration, and for each transport equation, an algebraic system of linear equations is solved iteratively until a prescribed residual decay is achieved.

All numerical schemes used to discretize the transport equations (convection schemes, diffusion, gradients, non-orthogonality corrections, eccentricity corrections) apply their low-order contributions implicitly, to the left-hand side of the equation system; and their higher-order contributions explicitly, to the right-hand side of the system, using values from the previous outer loop.

The code is parallelized using MPI and sub-domain decomposition. The grids are partitioned in sub-domains, each one having a layer of common cells so-called ghost-cells. Each of these sub-domains is calculated in its own MPI process. The ghost-cells are treated as normal cells, as far as the numerical algorithms are concerned, and are therefore handled implicitly.

ReFRESCO is currently being developed at MARIN (Netherlands) [8] in collaboration with IST (Portugal) [22], the University of Sao Paulo (Brazil) [24], the Technical University of Delft (the Netherlands) [16], the University of Groningen (the Netherlands) [4] and recently at the University of Southampton (UK) [12].

** corresponding author's e-mail*: J.Hawkes@soton.ac.uk

3 Strong Scalability Investigation

In previous work [12], ReFRESCO was profiled using *Score-P* [28] to extract timings from the code and its relevant functions. In a CFD code, these functions can be grouped together into the following categories, giving a breakdown of the code (see figure 1.a for illustration):

- **Assembly** – the assembly of each inner-loop linear equation system, including discretization, linearization and face-value interpolation for each transport equation.
- **Solve** – the iterative solving of the linear equation system for each transport equation, in each outer loop.
- **Gradients** – the calculation of gradient values at cell centres, using Gauss theorem.
- **Exchange** – MPI data exchange of cell-centred variables and gradients across ghost cells.
- **Other** – the remainder of the above, including initialization routines.

The results of a typical breakdown, using the well-known KVLCC2 case[2], are shown in figure 1, where the "speedup" is defined as the relative decrease in wall-time with N cores compared to the serial or single-node (16 cores) runtime. The results of the nodal speedup (1.b) show relatively good scalability (up to ≈40k cells-per-core), but hide the intra-node inefficiencies which are important for next-generation machines (with high intra-node parallelization). Graphs which show the speed-up relative to the single-core run-time (1.c) are thus more useful.

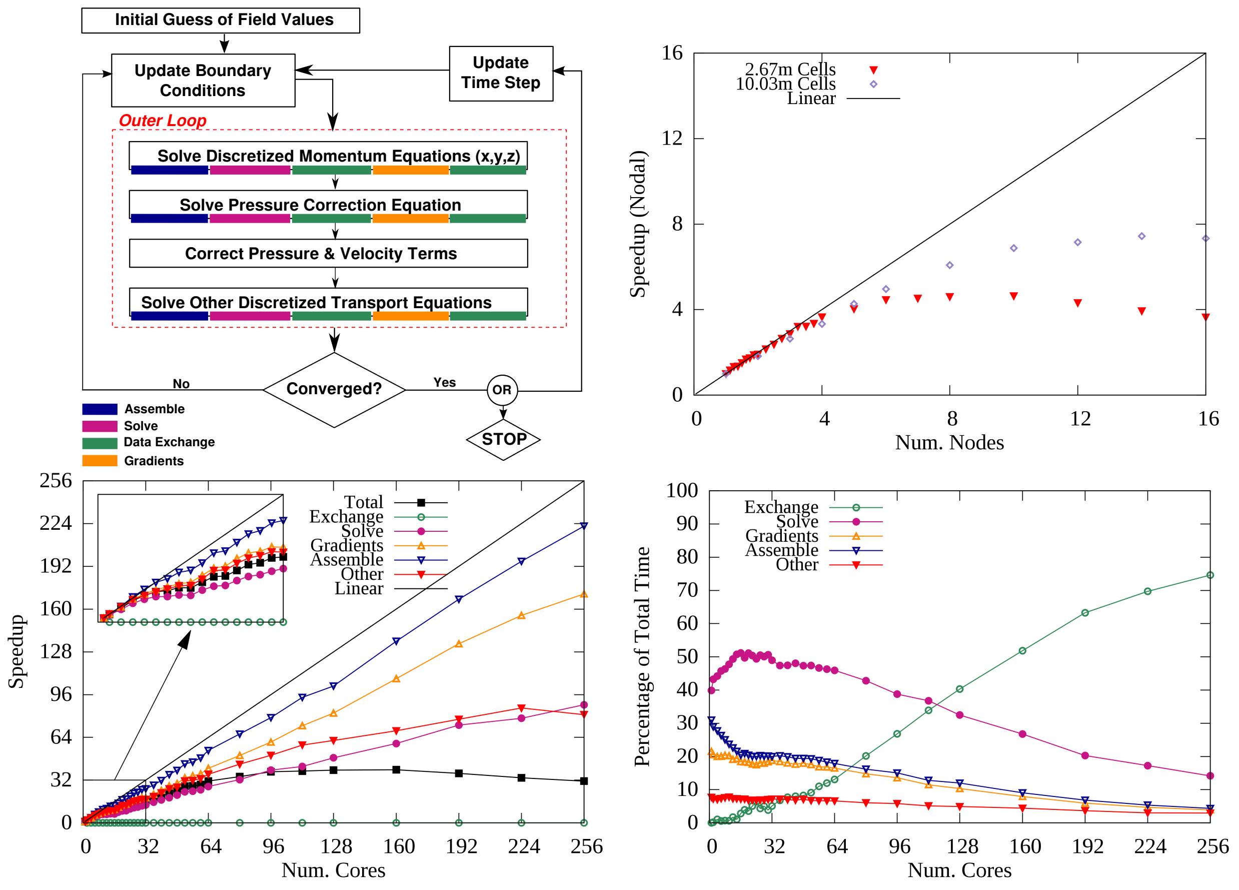


Figure 1: (a) An illustration of the SIMPLE algorithm, with colour-coding relating the various profiled functions of the code. (b) Total scalability of ReFRESCO normalized to single-node runtime, with two different grid sizes. (c) Scalability breakdown of a typical simulation showing the profiled functions individually and (d) the proportions of execution time spent in those routines [12].

The scalability graph (1.c) shows that *assembly* and *gradient* computations scale well, and the relative proportions graph (1.d) shows that these routines account for a very small proportion of run-time. The *other* routines do not scale so well, but are a small contribution to total run-time, so are of little concern.

The *solve* routines are an area for improvement. They exhibit poor scalability and take considerable amounts of total runtime, with particularly poor performance at the shared-memory level due to memory bandwidth or latency. Similarly, the *exchange* routines, which exhibit inverse scalability, are also a concern at the distributed-memory level. The data exchange is performed using MPI functions, and possibilities to improve this include switching to a hierarchical parallelization scheme (*i.e.* MPI + OpenMP) – as in [9, 10].

Up to ≈24k cells-per-core the iterative solver is the main limitation to scalability and should form the focus of future work, particularly as shared-memory parallelization grows. However, the data exchange issues are also an interesting area for further research and should not be neglected.

[2]KRISO Very Large Crude Carrier 2: half-body; two-equation κ-ω shear-stress transport (SST) turbulence model [21]; single-phase; based on wind-tunnel experiments [18]; 1000 outer loops; 2.67m structured grid.

4 Background to Iterative Solvers

The results shown in figure 1 used a Krylov subspace solver (GMRES - Generalized Minimal Residual method) with a Block Jacobi pre-conditioner [3]. Other Krylov methods such as BiCGStab (Bi-Conjugate Gradient Squared Stabilized) were tested with similar results. Other pre-conditioners were also tested, but Block Jacobi was (by far) the most scalable [12].

The Krylov solvers are powerful, but create a bottleneck due to the computation of inner products, which require global communication and synchronization in the form of MPI reductions. Efforts have been made to reduce the synchronization penalty of the Krylov solvers (down from two synchronized reductions to one, per iteration), with considerable improvements, but the bottleneck remains [20, 29].

By returning to simple, so-called stationary methods, it may be possible to obtain better performance in the limits of strong scalability. The task of a stationary solver (or, indeed, any iterative solver), for each transport equation in each outer loop, is to solve $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is the unknown solution vector, $\mathbf{A}$ is an n-by-n sparse coefficient matrix, $\mathbf{b}$ is the constant right-hand-side (RHS) vector, and n is the number of elements.

Beginning with an initial guess for $\mathbf{x}$, the system can be solved iteratively:

$$\mathbf{x}^k = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{k-1} + \mathbf{D}^{-1}\mathbf{b} \tag{1}$$

where $\mathbf{D}$ is the diagonal of $\mathbf{A}$, and $\mathbf{L}/\mathbf{U}$ are the lower- and upper-triangles respectively. The notation k represents the iteration number. This is the Jacobi method, and the matrix $\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ is the iteration matrix, $\mathbf{M}$. Each equation (from 1 to n) can be solved (*a.k.a.* relaxed) independently as follows:

$$x_i^k = \left(-\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{k-1} + b_i \right) / a_{ii}, \quad i = 1, \ldots, n. \tag{2}$$

where a, x and b are the individual components of $\mathbf{A}$, $\mathbf{x}$ and $\mathbf{b}$ respectively. At the end of each iteration the new values of $\mathbf{x}$ must be globally communicated before the next iteration can begin.

See Barrett et al. [5] for more information on a variety of iterative solvers – Krylov, stationary and otherwise.

5 Chaotic Iterative Solver

In 1969, Chazan & Miranker [7] proposed the concept of *Chaotic Relaxation* whereby several processes (distributed-memory processes or shared-memory threads) never synchronize. Instead, the processes freely pull values of off-diagonal $\mathbf{x}$ from memory whenever they are required, and push new values for the diagonal x_i whenever they have been relaxed. In this way, each relaxation uses the values of $\mathbf{x}$ from the latest iteration that is available – this could be several iterations behind the current relaxation iteration (s = 1,. . . ,n), or even ahead of it ($s < 0$):

$$x_i^k = \left(-\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{k-s} + b_i \right) / a_{ii}, \quad s < s_{max}, \quad i = 1, \ldots, n. \tag{3}$$

The order in which the n equations are relaxed is completely arbitrary. With this scheme, processes never need to wait for each other. Although communication must still occur, it can be entirely *asynchronous*, thereby making efficient use of memory bandwidth and computational resources. Processes may even iterate multiple times on the same data if memory bandwidth is completely saturated, making the best use of the available hardware. Whilst this method is based on the stationary Jacobi method, s can vary between iterations implying that chaotic methods are actually non-stationary.

Chazan & Miranker proved that this iterative scheme will converge for any real iteration matrix if $\rho(|\mathbf{M}|) < 1$ so long as s_{max} is bounded. $\rho(\cdot)$ denotes the *spectral radius* (the absolute value of the maximum eigenvalue) and $|\cdot|$ represents a matrix where all the components have been replaced with their absolute values. The implications of s_{max} being bounded is simply that if two relaxations take different amounts of time (either due to imbalanced hardware or relaxation complexity), they cannot be left completely independent indefinitely, such that s could potentially become infinite. Baudet [6] went on to prove that $\rho(|\mathbf{M}|) < 1$ is a necessary condition for convergence for any $s_{max} \leq k$. Baudet denoted the relaxation method where $s = 0, ..., k$ as an *asynchronous method* but the terms "chaotic" and "asynchronous" are often used interchangeably. Bahi [2] further showed that $\rho(|\mathbf{M}|) = 1$ is also valid, if $\mathbf{M}$ is singular and s_{max} is bounded.

Preconditioning of $\mathbf{A}$ usually serves to reduce the condition number and spectral radius of the equation system; however, preconditioning is rarely applied to simpler solvers, since almost all preconditioners are more complex than the solver itself.

At their conception, chaotic methods were considerably ahead of their time. Although created specifically for parallel computing, the concurrency of state-of-the-art supercomputers in 1969 was too small to utilize the methods efficiently. With new architectures, the true potential of chaotic methods may be realized. Anzt et al. [1] begins to show the use of chaotic or asynchronous methods on a modern architecture, using the GPU to perform block-relaxations. Despite the intrinsic loss in global convergence rates, Anzt et al. showed that chaotic iterative methods provided a boost in real-time convergence rates compared to standard stationary methods (*i.e.* Jacobi) – although comparisons to the more advanced Krylov methods were not performed, and the chosen matrices were not derived from CFD applications.

Chaotic methods provide a means to exploit massive parallelism due to the absence of synchronization points. They are also implicitly heterogeneous, allowing seamless cooperation between CPUs, GPUs or co-processors running at different speeds – allowing all computational resources to be used to maximum capacity with little concern for load-balancing. The convergence criteria, $\rho(|\mathbf{M}|) < 1$, is stricter than that of standard stationary methods which only require $\rho(\mathbf{M}) < 1$, and stricter still than the oft-used Krylov methods (which have no such requirements). The following section aims to determine if a range of standard CFD test cases will produce matrices that satisfy the criteria – the results of which will determine whether chaotic solvers are worth implementing and investigating.

6 Suitability of CFD Equation Systems

In this section, un-preconditioned matrices are extracted from a number of test cases. The matrices are analyzed to obtain key statistics and determine their suitability to chaotic methods.

For each matrix, it is possible to plot the connectivity graph, following the methods of Hu [15] – this gives a qualitative, visual insight into the sparse matrices by graphically connecting the elements of the matrix. The largest eigenvalues of $|\mathbf{M}|$ can be found using ARPACK [19] routines, and plotted in an Argand diagram – for $\rho(|\mathbf{M}|) < 1$ all eigenvalues must lie within a unit circle. The sparsity pattern of the matrix $\mathbf{A}$ may also be plotted directly to give qualitative clues on the matrix structure.

Assuming a satisfactory spectral radius, the *condition number* (the ratio of the largest to smallest eigenvalue) of the original matrix $\mathbf{A}$ can be used to assess the difficulty of convergence ($\propto$ number of iterations required). The condition number is computed using a 1-norm condition estimator [11].

As explained in section 2, all higher-order influences on the linear equation system are shifted to the RHS (the $\mathbf{b}$ vector) – which strongly decouples many common user settings (such as discretization scheme) from the format of the iteration matrix $\mathbf{M}$.

Changes in element-count and geometry may have a more profound effect on the matrices, and special equations (such as volume-fraction and cavitation equations) should also be examined. Thus the following test cases are chosen for increasing geometric complexity and their additional equations.

- **Lid-Driven Cavity Flow** (LDCF) on a number of 2D structured grids (225, 14.4k, 1m elements); no turbulence model; see [16].
- **NACA0015** hydrofoil (15° angle-of-attack) on a two-dimensional multi-block structured grid (28k elements) with a two-eqn. κ-ω SST turbulence model [21]; see [23].
- **KVLCC2** (half-body, no free surface, single-phase) on a three-dimensional multi-block structured grid (317k elements) and a hexahedral unstructured grid (**U**) with hanging nodes (167k elements); κ-ω SST turbulence; see [12].
- **NACA0015(C)**, as before, but with a Sauer-modified cavitation model; κ-ω SST turbulence; see [13].
- **Dambreak**, a homogeneous two-phase, three-dimensional problem with a simple structured grid (16k cells) with a volume-fraction equation; no turbulence; see [27].
- **Cylinder**, low Reynold's number, unsteady (10 timesteps) on a structured grid (4.3k elements); no turbulence; poor initial flow estimation; see [24].

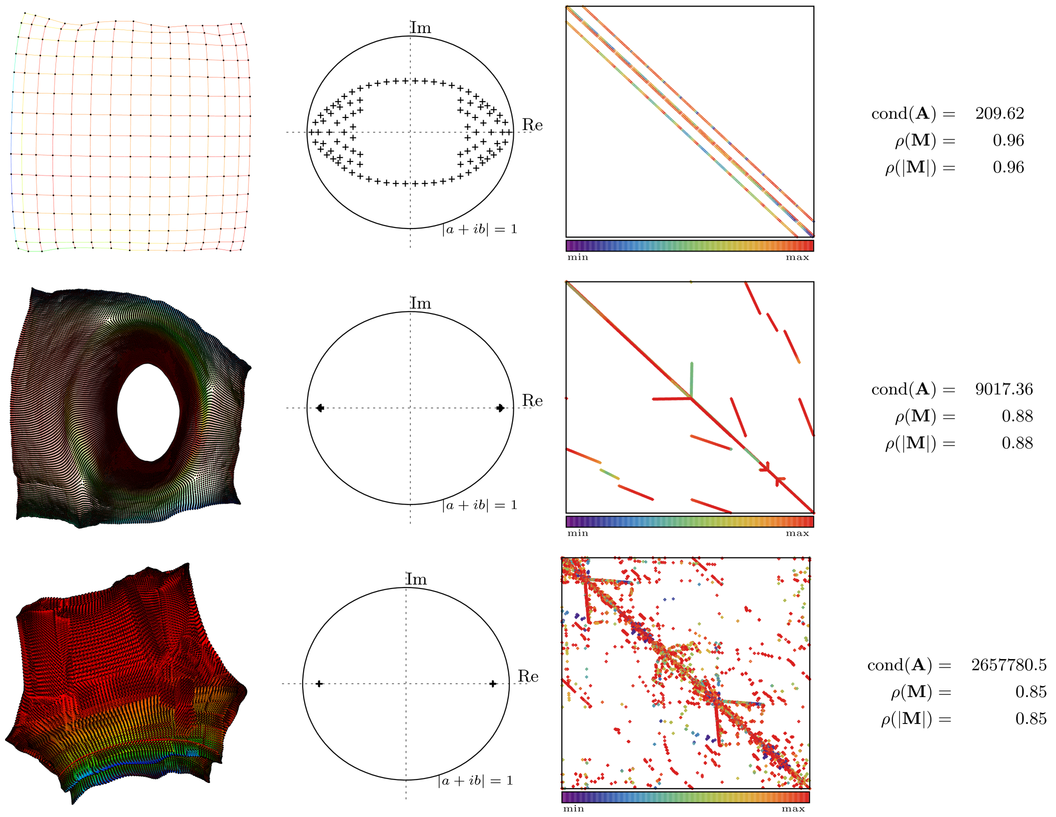
The matrices were extracted at outer loops 1, 5, 10, 50 and 100. The momentum equations (in x, y and z) are identical, due to the way in which they are assembled. The differences between $\rho(|\mathbf{M}|)$ and $\rho(\mathbf{M})$ were negligible, implying that $\mathbf{M}$ is mostly positive.

Figure 2 shows the qualitative view of the simple 2D LDCF, the more complex 2D hydrofoil, and the 3D KVLCC2 case at the 5^{th} outer loop. The connectivity graphs show resemblance to the underlying geometry and mesh structure – for example, the NACA0015 connectivity resembles the O-grid from which it arose.

The sparsity patterns are interesting from a computational perspective, since they highlight communication patterns when the matrix is split into parallel blocks. Where off-diagonal components of an equation are spread out, more cross-communication between processes is required – since the variables will be stored in parts of memory not directly accessible. The sparsity is closely related to the structure of the grid and the cell-numbering – the more complex KVLCC2 case is highly complex compared to the cartesian, structured LDCF grid.

Table 1 shows the quantitative results from all the test cases, taking the maximum values of all the extracted outer loops. The condition number appeared to be higher for more complex cases, such as the 3D KVLCC2 case, although there was little correlation with flow features or mesh structure. In all cases, $\rho(|\mathbf{M}|) \leq 1$, meeting the requirements for chaotic solvers.

The pressure equation, which takes a Poisson-equation format, was singular for the LDCF and Dambreak case, where only Neumann boundary conditions are applied. In all the other cases, a Dirichlet condition on the outflow reduced the spectral radius, although it was still close to one. In the LDCF-225 case and the first timestep of the unsteady cylinder, the spectral radius was also very high – these cases both feature complex flows with simple initial-flow estimations and coarse meshes; this large discrepancy and poor resolution could be the reason for the the near-singular matrices (however, the condition number was still low).



$\mathrm{cond}(\mathbf{A}) =$ 209.62
$\rho(\mathbf{M}) =$ 0.96
$\rho(|\mathbf{M}|) =$ 0.96

$\mathrm{cond}(\mathbf{A}) =$ 9017.36
$\rho(\mathbf{M}) =$ 0.88
$\rho(|\mathbf{M}|) =$ 0.88

$\mathrm{cond}(\mathbf{A}) =$ 2657780.5
$\rho(\mathbf{M}) =$ 0.85
$\rho(|\mathbf{M}|) =$ 0.85

Figure 2: Connectivity, 100 largest eigenvalues, sparsity and statistics for the momentum equation (outer loop 5). [Top] LDCF-225, [Middle] NACA0015-28k and [Bottom] KVLCC2-317k.

Additional matrices for the KVLCC2 case (up to 2.67m elements) and a 21.7m-element INSEAN E779A propeller (with sliding interface) were tested, but could not be fully post-processed due to memory requirements – nonetheless, the original matrices ($\mathbf{A}$) were diagonally dominant: a sufficient condition for $\rho(|\mathbf{M}|) < 1$.

Table 1: Quantitative results for a range of test matrices, showing condition number and spectral radius for momentum-, pressure-, turbulence- and free-surface-/cavitation-equations.

	Mom.		Pres.		Turb. 1		Turb. 2		V.F./Cav.	
	$\mathrm{cond}(\mathbf{A})$	$\rho(\lvert\mathbf{M}\rvert)$	$\mathrm{cond}(\mathbf{A})$	$\rho(\lvert\mathbf{M}\rvert)$	$\mathrm{cond}(\mathbf{A})$	$\rho(\lvert\mathbf{M}\rvert)$	$\mathrm{cond}(\mathbf{A})$	$\rho(\lvert\mathbf{M}\rvert)$	$\mathrm{cond}(\mathbf{A})$	$\rho(\lvert\mathbf{M}\rvert)$
LDCF-225	210	0.955	9.4·e17	1.000						
LDCF-14.4k	7	0.500	2.1·e18	1.000						
LDCF-1m	4	0.500	2.9·e19	1.000						
NACA0015	9.0·e3	0.884	6.53·e6	0.999	4.5·e3	0.849	4.8·e3	0.851		
NACA0015(C)	6.3·e3	0.734	4.67·e6	0.999	5.1·e3	0.688	5.2·e3	0.690	2.3·e8	0.240
KVLCC2	2.6·e6	0.845	2.63·e8	.9999	1.2·e7	0.588	1.2·e7	0.594		
KVLCC2 (unst.)	5.2·e6	0.810	5.4·e7	.9999	8.3·e6	0.726	7.7·e6	0.728		
Dambreak	105	0.048	5.6·e18	1.000					7	0.048
Cylinder (t=1)	111	.9999	1.1·e6	.9999						
Cylinder (t=2–10)	4.4·e2	0.520	1.1·e6	.9999						

7 Conclusion

Chaotic iterative solvers have been proposed as a means to improve the scalability of a typical CFD code, in preparation for a paradigm-shift towards massive parallelization in supercomputing architectures. The removal of synchronization points from one of the major bottlenecks in the code (the *solve* routines) could lead to a vast improvement in scalability, although the loss in convergence rate which compromises the speed-up is difficult to predict. The matrices which the chaotic solver would be required to solve have been extracted and evaluated, and it has been determined that necessary and sufficient conditions for convergence have been satisfied.

Future work will focus on implementing a chaotic solver in a hierarchical parallel environment (MPI & OpenMP & GPU/coprocessor). At the shared-memory (OpenMP) level, issues associated with double-precision atomic operations and intra-core cache-coherency will be encountered; and methods must be investigated to hide any cache latency that may occur. Maintaining true asynchronicity across memory-boundaries (for example, across nodes, using MPI) will also be a unique challenge – since most MPI communications require at least two nodes to synchronize. One-sided MPI operations may be investigated to allow direct access to non-shared memory. Similarly, true asynchronicity between host processes and attached GPUs/coprocessors must be achieved – perhaps by assigning processes on the host processor purely for these communications.

From a numerical perspective, it is best if each equation in the system is iterated the same number of times/at the same speed (*i.e.* the equations do not become too out-of-synch). Regulating this, by moving equations to different processes dynamically, may be beneficial for overall performance and may be necessary for convergence when the pressure equation is singular.

Above all, however, is the intrinsic difficulty of debugging a chaotic scheme. As soon as instrumentation points are added, the chaotic order of the process changes – indeed, every time the process is run, different numerical results and convergence rates may be obtained.

Chaotic methods introduce a number of unique challenges, but could create a significant speed-up for CFD on current and next-generation supercomputers. Whilst it has been shown that the chaotic methods will converge for the relevant equations, it remains to predict their real-world performance for CFD applications; and to implement and test them successfully.

References

[1] Anzt, H., Tomov, S., Dongarra, J. & Heuveline, V. (2013, December). A Block-Asynchronous Relaxation Method for Graphics Processing Units. *Journal of Parallel and Distributed Computing*, 73(12):1613–1626.

[2] Bahi, J. (2000). Asynchronous Iterative Algorithms for Nonexpansive Linear Systems. *Journal of Parallel and Distributed Computing*, 60:92–112.

[3] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F. & Zhang, H. (2013). PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory. `http://www.mcs.anl.gov/petsc`.

[4] Bandringa, H., Verstappen, R., Wubbs, F., Klaij, C. & Ploeg, A. (2012, October). On Novel Simulation Methods for Complex Flows in Maritime Applications. *Numerical Towing Tank Symposium (NUTTS)*, Cortona, Italy.

[5] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. & der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA.

[6] Baudet, G. M. (1978, April). Asynchronous Iterative Methods for Multiprocessors. *J. ACM*, 25(2):226–244.

[7] Chazan, D. & Miranker, W. (1969, April). Chaotic Relaxation. *Linear Algebra and its Applications*, 2(2):199–222.

[8] Eca, L. & Hoekstra, M. (2012, August). Verification and Validation for Marine Applications of CFD. *29th Symposium on Naval Hydrodynamics*, Gothenburg, Sweden.

[9] Gorobets, A., Trias, F. & Oliva, A. (2013, December). A Parallel MPI+OpenMP+OpenCL Algorithm for Hybrid Supercomputations of Incompressible Flows. *Computers & Fluids*, 88:764–772.

[10] Gropp, W. D., Kaushik, D. K., Keyes, D. E. & Smith, B. F. (2001, March). High-Performance Parallel Implicit CFD. *Parallel Computing*, 27(4):337–362.

[11] Hager, W. W. (1984). Condition Estimates. *SIAM Journal on Scientific and Statistical Computing*, 5:311–316.

[12] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B. & Vaz, G. (2014, October). Performance Analysis Of Massively-Parallel Computational Fluid Dynamics. Submitted to *The 11th International Conference on Hydrodynamics (ICHD)*, Singapore.

[13] Hoekstra, M. & Vaz, G. (2009, August). The Partial Cavity on a 2D Foil Revisited. In proceedings of the *7th International Symposium on Cavitation*, Ann Arbor, MI, USA.

[14] Horst, S. (2013, May). Why We Need Exascale And Why We Won't Get There By 2020. *Optical Interconnects Conference*, Santa Fe, New Mexico, USA.

[15] Hu, Y. (2005). Efficient, High-Quality Force-Directed Graph Drawing. *Mathematica Journal*, 10(1):37–71.

[16] Klaij, C. & Vuik, C. (2013). Simple-Type Preconditioners for Cell-centered, Collocated, Finite Volume Discretization of Incompressible Reynolds-averaged Navier-Stokes Equations. *International Journal for Numerical Methods in Fluids*, 71(7):830–849.

[17] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, S. & Yelick, K. (2008, September). ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. *DARPA IPTO*.

[18] Lee, S., Kim, H., Kim, W. & Van, S. (2003). Wind Tunnel Tests on Flow Characteristics of the KRISO 3,600 TEU Container Ship and 300K VLCC Double-Deck Ship Models. *Journal of Ship Research*, 47(1):24–38.

[19] Lehoucq, R. B., Sorensen, D. C. & Yang, C. (1998). *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems by Implicitely Restarted Arnoldi Methods.* SIAM, Philadelphia, PA, USA.

[20] Maheswaran, M., Webb, K. & Siegel, H. (1998, August). Reducing the Synchronization Overhead in Parallel Nonsymmetric Krylov Algorithms on MIMD Machines. In proceedings of *International Conference on Parallel Processing*, Minneapolis, Minnesota, USA.

[21] Menter, F., Kuntz, M. & Langtry, R. (2003, October). Ten Years of Industrial Experience with the SST Turbulence Model. In *Turbulence, Heat and Mass Transfer 4*. Antalya, Turkey.

[22] Pereira, F., Eca, L. & Vaz, G. (2013, June). On the Order of Grid Convergence of the Hybrid Convection Scheme for RANS Codes. In proceedings of *CMNI*, Bilbao, Spain.

[23] Rijpkema, D. R. (2008, November). Numerical Simulation of Single-Phase and Multi-Phase Flow over a NACA 0015 Hydrofoil. M.Sc. Thesis, Delft University of Technology, Faculty of 3ME.

[24] Rosetti, G., Vaz, G. & Fujarra, A. (2012, July). URANS Calculations for Smooth Circular Cylinder Flow in a Wide Range of Reynolds Numbers: Solution Verification and Validation. *Journal of Fluids Engineering, ASME*, page 549.

[25] Shalf, J. (2013, October). The Evolution of Programming Models in Response to Energy Efficiency Constraints. *Oklahoma Supercomputing Symposium*, Norman, Oklahoma, USA.

[26] Top 500 List (Acc. 2013, February). `http://www.top500.org`.

[27] Vaz, G., Jaouen, F. & Hoekstra, M. (2009, May–June). Free-Surface Viscous Flow Computations: Validation of URANS Code FRESCO. *28th International Conference on Ocean, Offshore and Arctic Engineering (OMAE)*, Honolulu, Hawaii.

[28] VI-HPS, Score-P, v.1.2.3 (Acc. 2013, November). `http://www.vi-hps.org/projects/score-p`.

[29] Zuo, X.-Y., Zhang, L.-T. & Gu, T.-X. (2014, December). An Improved Generalized Conjugate Residual Squared Algorithm Suitable for Distributed Parallel Computing. *Journal of Computational and Applied Mathematics*, 271:285–294.

VI International Conference on Computational Methods in Marine Engineering
MARINE 2015
F. Salvatore, R. Broglia and R. Muscari (Eds)

CHAOTIC LINEAR EQUATION-SYSTEM SOLVERS FOR UNSTEADY CFD

JAMES N. HAWKES[1,2], STEPHEN R. TURNOCK[1], GUILHERME VAZ[2], SIMON J. COX[1] AND ALEX B. PHILLIPS[3]

University of Southampton [1]
Boldrewood Campus, Southampton, United Kingdom
e-mail: J.Hawkes@soton.ac.uk

Maritime Institute Netherlands (MARIN) [2]
Wageningen, Netherlands

National Marine Facilities [3]
National Oceanography Centre, United Kingdom

Key words: Computational Methods, Computational Fluid Dynamics, Linear Solvers, Totally Asynchronous Methods, Chaotic Iterative Methods, PETSc, ReFRESCO.

Abstract. A Chaotic Iterative Method, which is a form of totally asynchronous linear equation-system solver, is implemented within an open-source framework. The solver is similar to simple Jacobi or Gauss-Seidel methods, but is highly optimized for massively-parallel computations. Processes or threads are free to run computations regardless of the current state of other processes, iterating individual equations with no limitations on the state of the variables which they use. Each individual iteration may pull variables from the same iteration, the previous iteration, or indeed any iteration. This effectively removes all synchronization from the Jacobi or Gauss-Seidel algorithm, allowing computations to run efficiently with high concurrency.

The trade-off is that the numerical convergence rate of these simple algorithms is slower compared to the classical Krylov Subspace methods, which are popular today. However, unique features of the computational fluid dynamics algorithm work in favour of Chaotic methods, allowing the fluid dynamics field to exploit these algorithms when other's cannot.

The results of the Chaotic solver are presented, verifying the numerical results and benchmarking performance against the Generalized Minimal Residual (GMRES) solver and a Pipelined GMRES solver. The results show that, under certain circumstances, Chaotic methods could be used as a standalone solver due to their superior scalability. The potential to use Chaotic methods as a pre-conditioner or hybrid solver is also revealed.

1 INTRODUCTION

One of the major objectives in high performance computing for computational fluid dynamics is to enable complex unsteady simulations. Whilst these computations benefit from concurrency in three spatial dimensions, time-stepping is strictly serial; and unsteady computations become bottle-necked by the amount of parallelism that can be utilized in the spatial dimensions. Combined with this, the architecture of modern supercomputers is changing rapidly, causing scientific applications to adapt to a many-core era.

Figure 1.a shows the history of the Top 500 supercomputers over the last 20 years. At current rates, it is expected that the first computer capable of 10^{18} floating-point-operations-per-second (1 ExaFLOPS) could be built in 2020. Its capability will be limited by electrical power consumption, rather than pure hardware speed, and this will lead to two significant architecture changes [9, 11, 14]. Firstly, core clock rates must decrease to save power, and total computing power must be improved by dramatically increasing the total number of cores. Secondly, inter-nodal networks must be reduced due to their relatively large power consumption. Overall, this means a huge growth in concurrency within nodes. This trend can already be seen in modern 'many-core' systems featuring GPUs or co-processors.

The current most powerful computer, Tianhe-2 [15], capable of 33 PFlops, consists of 3,120,000 cores. By contrast, the first exascale machine will be 30-times more powerful and 300-times more parallel. Whilst Tianhe-2 *already* consists of nodes with O(100) cores, using co-processors; an exascale machine is likely to feature O(1k-10k) cores per node.

In earlier work [6] the strong scalability of a typical CFD code (ReFRESCO) was assessed, to discover which parts of the algorithm required improvement for this paradigm-shift in architecture. Figure 1.b shows the breakdown of the code, which is based on the SIMPLE algorithm, into its key parts. Figure 1.c shows that the *assembly* (discretization, orthogonality & eccentricity corrections, and final assembly of the matrix structure) and *gradient computations* scale well, and figure 1.d shows that these routines account for a small proportion of the overall run-time. The *solve* routines, responsible for solving the linear-equation systems that have been assembled, exhibit poor scalability and take considerable amounts of total runtime. Similarly, the *exchange* routines (responsible for sharing of cell-values and gradient values across ghost cells) are a cause for concern, dominating total run-time when inter-nodal communication becomes large.

Considerable work is being performed to improve the *exchange* routines, primarily by overlapping communication with computation more efficiently or reducing global communications; and by utilizing a hybrid parallelization scheme. Cooperatively, the *solve* routines must be improved with the same goals.

In this paper, a 'Chaotic' iterative solver is proposed as a solution to the strong scalability problem of the *solve* routines. In the following sections, a brief background is given, followed by the theory and implementation of a Chaotic solver. Finally, the Chaotic solver is verified and benchmarked against existing solvers.

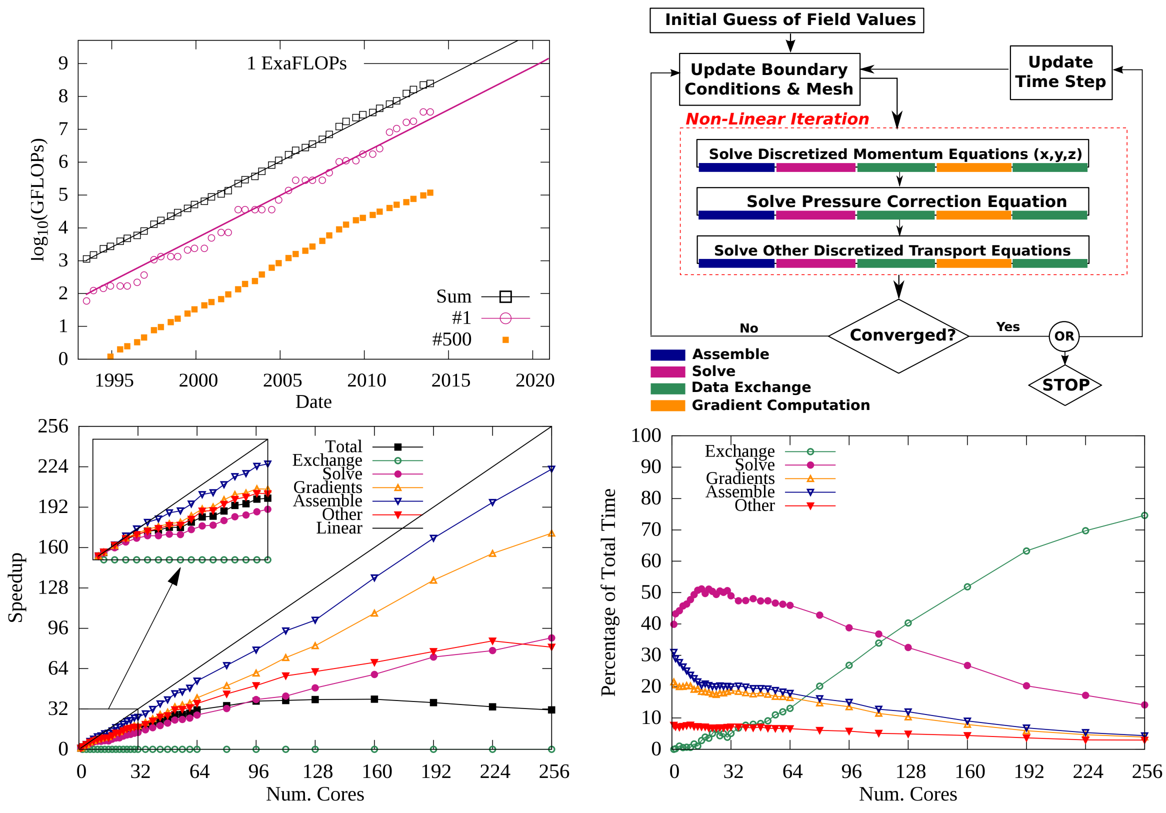


Figure 1: (a) Total floating-point operation (FLOP) rate of the Top500 supercomputers, showing the #1 machine, the #500, and the sum of the top 500. (b) An illustration of the SIMPLE algorithm, with colour-coding relating the various sections of the code. (c) Scalability breakdown of a typical simulation showing the parallel speed-up of the code compared to its serial operation and (d) the proportions of execution time spent in the various routines [6].

2 BACKGROUND TO LINEAR EQUATION-SYSTEM SOLVERS

The task of the linear equation-system solver, for each transport equation in each non-linear loop, is to solve $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is the unknown solution vector, $\mathbf{A}$ is an n-by-n sparse coefficient matrix created by the *assembly* routines, $\mathbf{b}$ is the constant right-hand-side (RHS) vector, and n is the number of elements.

It is possible to solve this system directly, but for large matrices this is expensive. For CFD, the equation systems can be solved iteratively:

$$\mathbf{x}^k = -\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}^{k-1} + \mathbf{D}^{-1}\mathbf{b} \tag{1}$$

where $\mathbf{D}$ is the diagonal of $\mathbf{A}$, and $\mathbf{L}/\mathbf{U}$ are the lower- and upper-triangles respectively. k is the iteration number. This is the Jacobi method, and the matrix $\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})$ is the iteration matrix, $\mathbf{M}$.

Numerically, the Jacobi method, or its derivatives such as Gauss-Seidel or Successive Over-Relaxation methods, are weak. Their convergence rate is limited by the spectral radius of the iteration matrix, $\rho(\mathbf{M})$, which is dependent on the matrix size and stiffness. For CFD, this means the pressure equation is hardest to solve, as it is close to singular [7] and the spectral radius is close to unity.

It is possible to add a form of 'memory' to these solvers, where some information about the solution path is stored in a vector subspace, known as the Krylov Subspace (KSP). For example, in the popular (Bi-)Conjugate Gradient Squared (BCGS) algorithm, the solver remembers the previous gradients that it followed, to prevent it from going back on itself. In the GMRES (Generalized Minimal Residual Method) the subspace is based on the Arnoldi iteration, and is used to find an approximate solution based on the minimal residual of these vectors. See Barrett et al. [2] for more information on a variety of iterative solvers.

There is an initial grace period where high frequency errors can be reduced more quickly than the spectral radius would normally allow, using the simpler methods, but KSP solvers typically over-power the simpler solvers when lower frequencies must also converge.

Though numerically powerful, the KSP methods are computationally inefficient on a larger number of cores due to certain communication patterns. Figure 2 shows an illustration of a GMRES iteration. Two significant communication patterns are required, with very different scaling characteristics. Firstly, the sparse-matrix-vector-multiplication (SpMV) requires neighbour-to-neighbour communication to obtain boundary data from other processors. This communication pattern is highly scalable, as different neighbours can communicate in parallel.

The second communication pattern is a global reduction (and associated broadcast). Many small messages are sent down a hierarchical tree in a non-concurrent manner. The time for a reduction *increases* as the number of cores increases, whereas every other routine receives a speed-up from increasing the number of cores.

In the GMRES or BCGS algorithm there are two global reductions per iteration. As the number of cores rises and the scalable routines reduce to a small proportion of overall time, these reductions dominate wall-time and prevent the KSP solvers from scaling. Improvements have been made to GMRES, in the form of 'pipelined' GMRES (P-GMRES) which attempt to overlap the first reduction with serial computations [5] – results show better scalability than GMRES, especially when the so-called 'latency-factor' is large, but as the cells-per-core ratio decreases global reduction latency will once again dominate.

Compared to other applications, where solving linear equation systems to machine accuracy is important (for example, solving Markov chains for risk analysis), CFD is not so strict. As the 'real' problem is a non-linear one, much of the work done in the linear loops is discarded as under-relaxation is applied to the non-linear iteration. In fact, for many cases, it is possible to reach non-linear convergence with a relative convergence factor of the linear system as high as 60%, although this is not usually beneficial. It highly depends on the problem and the type of non-linear iterative process; and is only worthwhile if the non-linear iterative process is efficient, as many more non-linear iterations are required.

With this in mind, it may be possible to take advantage of the initial 'grace period' of the simpler solvers such as Jacobi or Gauss-Seidel; however, to compete with a well-

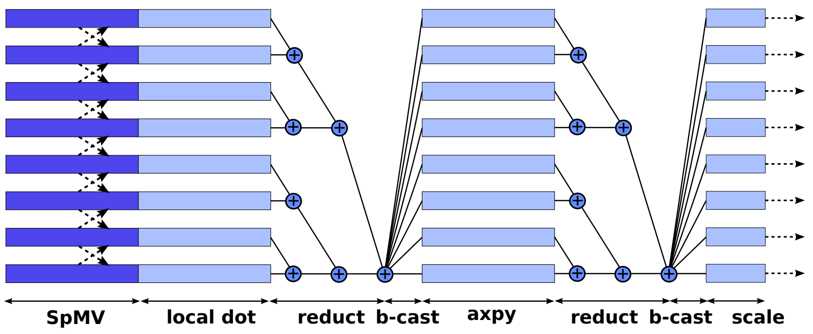


Figure 2: Illustrative parallel trace of a GMRES iteration, showing the computation and communication patterns when utilizing 8 cores. Note that as the serial computations and SpMV become small, the reduction dominates the total wall-time. Not to scale.

preconditioned GMRES solver there must be a significant computational benefit.

3 THEORY AND IMPLEMENTATION OF A CHAOTIC SOLVER

The Jacobi solver is particularly poor numerically, using old values of the solution vector in each iteration rather than using newly-computed variables from the current iteration. By contrast, Gauss-Seidel is stronger numerically as it uses newly-computed variables; but is impossible to parallelize. Forms of Block Gauss-Seidel exist, which are a good compromise; but both Jacobi and Block Gauss-Seidel can be improved computationally.

The Jacobi or Block Gauss-Seidel methods have no reduction routines, using purely an SpMV to perform the iterations. This is a huge benefit to scalability, and should allow these solvers to scale far higher than KSP methods. However, the Jacobi/Block G-S routine still has room for improvement. In these methods, the $k+1^{th}$ iteration cannot start until the k^{th} iteration is complete. This means that the whole solution can only proceed as fast as the slowest core. A core may be slow due to background operating-system load, poor load-balancing, heterogeneity in the hardware, irregular communication times or Non-Uniform Memory Access (NUMA). These irregularities are likely to be more of a concern in the many-core era, and will significantly slow down computations when trying to utilize thousands of cores.

In 1969, Chazan & Miranker [3] proposed the concept of *Chaotic Relaxation* whereby several processes never synchronize. Instead, the processes freely pull values of off-diagonal $\mathbf{x}$ from memory whenever they are required, and push new values for the diagonal x_i whenever they have been relaxed. By exploiting data races in this way, each relaxation uses the values of $\mathbf{x}$ from the latest iteration that is available – this could be several iterations behind ($s = 1$, ..., k), or even ahead of it ($s < 0$):

$$\mathbf{x}^k = -\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}^{k-s} + \mathbf{D}^{-1}\mathbf{b} \tag{2}$$

In the average case, this scheme should produce numerical convergence similar to that of Gauss-Seidel.

In previous work [7] it was shown that the strict convergence criteria for Chaotic methods are met by all CFD equations, which prompts the implementation and testing of a Chaotic solver, as follows.

The solver is written in a hybrid MPI + OpenMP (Open Multi-Processing – a multi-threading standard) environment, using PETSc (Portable, Extensible Toolkit for Scientific Computation) [1] as a platform. The MPI processes and OpenMP threads are split in a NUMA-friendly manner, with one MPI process per socket and one thread per core within each socket. The test machine is Iridis 4, the University of Southampton's latest super-computer, which consists of 750 compute nodes - each with 2 Intel Xeon E5-2670 Sandy-bridge processors (8 cores, 2.6 Ghz). Within each MPI process one thread is dedicated to halo-data communication and the remaining seven perform the chaotic matrix-vector multiplications.

In the following sections, the results of the Chaotic solver will be verified by comparing the solutions produced by them against KSP methods, and the performance (in terms of scalability and absolute wall-time) will be assessed.

4 VERIFICATION

The solution produced by the Chaotic solver was compared against the solution produced by GMRES (with a restart of 3, 30 and 100), P-GMRES and BCGS. Pre-conditioners were not used and there was no attempt to compare performance, these tests were performed purely to compare the absolute solution and verify the implementation. The solvers were required to iterate until a desired absolute residual in the l^2-norm.

Three analytical matrices from the matrix marketplace [4] were tested ($pde225$, $pde900$ and $orsirr\_1$), along with a pressure equation matrix extracted from a 14.4k-cell lid-driven cavity flow simulation ($LDCF$). The results for the smallest case ($pde225$) are shown in figure 3. The results were similar for all cases and a range of convergence tolerances from 10^{-1} to 10^{-7} (not shown).

There are no concerns for any of the solvers, although there are some differences in the solution vector. There is a tendency for the Chaotic solver to produce a much lower l^1-norm, presumably as the stiffer equations in the system require many iterations to solve, the easier equations can progress much more quickly, whereas the KSP methods have more inter-dependency between the equations.

Due to the squared-weighting of the l^2-norm this has little effect on the reliability of the solution, and differences in the overall flow field are likely to be negligible.

5 PERFORMANCE TESTING

The aim of Chaotic solvers is to improve scalability, and it is expected that at lower core-counts that KSP solvers will out-perform the Chaotic methods. In order to predict how these solvers will perform on many-core machines, scalability tests are the main focus of the following tests. It should be possible to predict the break-even point of Chaotic methods, where their worse numerical performance overtakes the KSP solvers based on improved scalability.

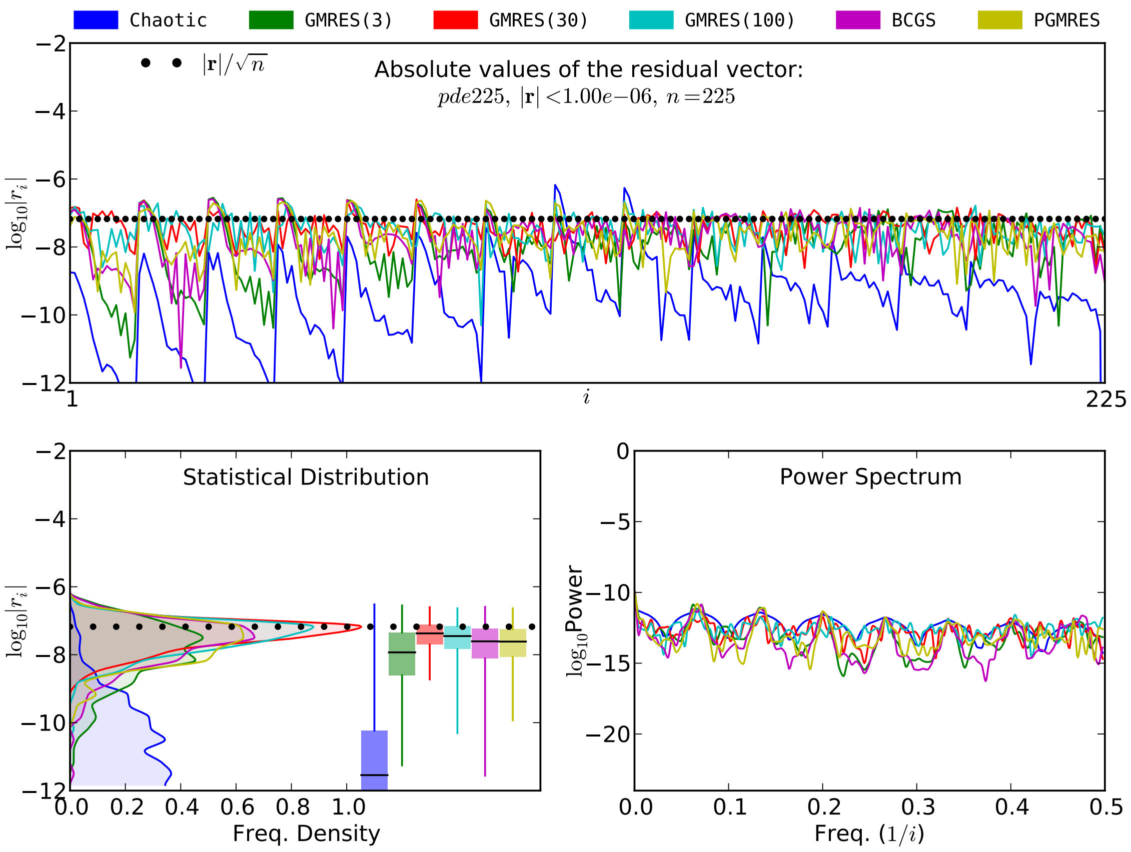


Figure 3: Full verification results from the *pde225* case, with a requested absolute l^2-norm of 10^{-6}. [Top] A plot of the raw residual vector ($\mathbf{r} = \mathbf{Ax} - \mathbf{b}$). [Bottom-Left] Statistical Distribution of the residual vector, showing the mean, upper/lower quartiles, and minimum/maximum values. [Bottom-Right] Spectral analysis of the residual vector, with peaks corresponding to frequencies of error in the converged system. The dotted lines in the results mark the *mean value of the residual required in order to satisfy the l^2-norm*. This line divides the elements of the residual that positively or negatively contribute to the l^2-norm.

A number of benchmark cases are used in order to capture a range of problem sizes:

- **KVLCC2 Tanker** (half-body, single-phase) on three-dimensional, multi-block structured grids (317k, 1.11m, 2.67m, 5.99m, 10.0m & 15.8m elements), using a two-equation κ-ω SST turbulence [12]. See [6].
- **Lid-Driven Cavity Flow** (LDCF) on 2D structured grids (14.4k & 1m elements). No turbulence model. See [10].
- **NACA0015C** hydrofoil (15° angle-of-attack) on a two-dimensional multi-block structured grid (28k elements), using a κ-ω SST turbulence model, and a Sauer-modified cavitation model. See [8].
- **Dambreak**, a homogeneous two-phase, three-dimensional problem with a simple structured grid (16k cells) and a volume-fraction equation. No turbulence model.

See [16].

- **Cylinder** at a low Reynold's number. An unsteady simulation (10 non-linear iterations per timestep) on a structured grid (4.3k elements) with a poor initial flow estimation. No turbulence model. See [13].

A large number of core-counts are tested, beginning from 32 cores, to which the scalability graphs are normalized:

$$Scalability = \frac{Time(\text{32-cores})}{Time(\text{n-cores})} \tag{3}$$

The linear system convergence tolerance is set to 60% and 50 non-linear iterations are performed. Chaotic solvers were compared against GMRES and P-GMRES, both with a restart of 30. The respective solver was applied to every equation (*i.e.* momentum, pressure, turbulence, volume-fraction and cavitation), and their average taken for each case.

It was necessary to use pre-conditioning on the GMRES solvers in order to realize their true potential. Unfortunately these solvers then base their convergence on the preconditioned residual $\mathbf{P^{-1}}(\mathbf{Ax} - \mathbf{b})$ which is not proportional to the true residual. If the pre-conditioner is not smooth, and the matrix is close to singular (as in the pressure equation) a 10% relative convergence tolerance in the preconditioned residual may be equivalent to a 50% or higher convergence tolerance in the true residual. To test fairly, the pre-conditioner was limited to a simple diagonal scaling (Jacobi) and the equivalent residual was computed in the Chaotic solver. This is not the best-case scenario for the KSP solver, as pre-conditioning can drastically change their performance. However, there is some overhead in the Chaotic solver in computing the diagonally-scaled residual, so it seems the best compromise. Regardless, the effects of pre-conditioning should not have a huge impact on the scalability results, especially given that Jacobi pre-conditioning is highly scalable compared to more complex pre-conditioners.

Figure 4 shows the performance results for each solver. Firstly, the tabular results show the absolute wall-time comparison on 32-cores and 256-cores. The Chaotic solvers are expectedly slower on all reasonably-sized test cases and up to 6.88 times slower than GMRES in the largest test case. The differences between GMRES and Chaotic solvers in terms of absolute time mostly depend on the convergence tolerance (tolerances of 40% and 20% were partially tested – not shown), however, scalability is mostly unchanged. P-GMRES performs approximately 50% slower where strong scalability is not a concern.

The scaling graphs require more careful analysis. There are three factors at play: (1) the cells-per-core ratio, which governs how much serial work and scalable ghost-cell exchange each process/thread performs; (2) the absolute number of processes, which governs how much time is spent performing reductions; and (3) cache effects. The combination of all three, and their relativity to each other, determines scalability.

The cells-per-core ratio is dominant in terms of scalability, since it changes most dramatically with the number of cores (doubling the number of cores halves the time due to factor (1)); whereas doubling the number of cores has little change on factor (2), as the reduction time increases at a rate of $\log_2(cores)$. Cache effects (3) occur when the main parts of an algorithm fit into a high-speed memory buffer, located on the CPU. The cache is responsible for the super-linear scalability demonstrated by each solver. Its effect is more qualitative, as one could do extensive cache-optimization for each solver and each CPU to obtain the best results. No such optimization was performed for these solvers, and one can make the argument that the Chaotic solver more naïvely exploits the cache, leading to performance benefits at the intra-node level, where memory contention is a bottleneck. For the Chaotic solver, it is possible to see the point where the working memory fits into cache ($\approx$20k-30k cells-per-core).

The scalability of GMRES and P-GMRES collapses when a certain cells-per-core ratio is reached - typically around 10k cells-per-core. Chaotic solvers continue to scale beyond this point. For example, with the KVLCC2-1.11m case on 512 cores, the scalability of the Chaotic solver is a factor of 601 compared to GMRES at 104. In absolute terms, the Chaotic solver is approximately twice as fast at $\approx$2.2k cells-per-core.

This scalability does not continue indefinitely, as demonstrated by the very small test cases. This is due to the expense of setting up the 8 OpenMP threads each time the solver is called, as well as a few other serial bottlenecks in the initialization, finalization and interface with PETSc's data structures; there is also some 'lag' involved with the residual computations which can cause over-convergence of very small equations. The reduction required to compute the residual also blocks the SpMV MPI communications, which is not ideal.

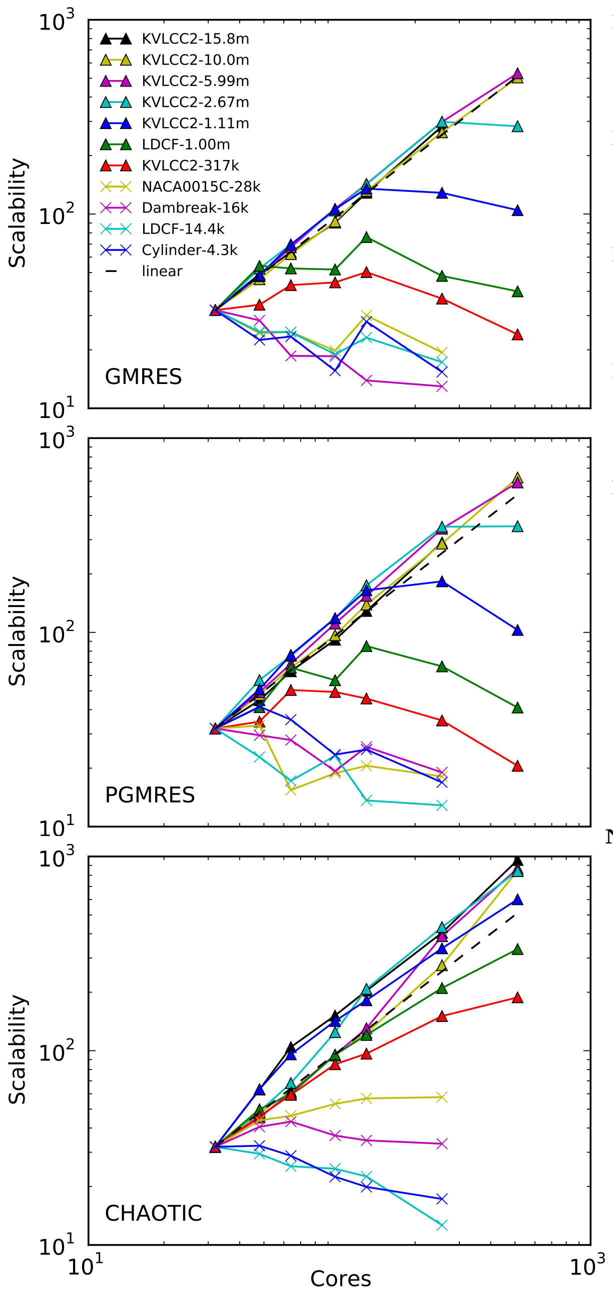
Some scalability is also lost due to the MPI communications themselves, which, although asynchronous with the computations, ultimately proceed in lock-step with each other. This means two MPI processes cannot benefit from faster communication between themselves.

The scalability of P-GMRES is lacklustre for this application. It is suspected that the very lax convergence tolerance prevents P-GMRES from showing its true benefit (since the number of total iterations is very small and more time is spent in setting up the solver), and very naïve pre-conditioning is used. It is also known that the scalability can be improved by increasing the l-factor of the solver, and this could be further investigated [5].

It is important to note that the results show the average of all the equation systems, although the pressure equation dominates the others in terms of total wall-time. Splitting the analysis between each equation individually is important, but the relative comparison between solvers is virtually unchanged in these cases.

6 CONCLUSION

The results show that the Chaotic solver provides a correct solution and that the scalability is better than a typical Krylov solver. The comparison of absolute wall-time is subjective - utilizing such high convergence tolerance of the linear equation system may not be viable, depending on the non-linear iterative process, the particular test case, and the efficiency of the non-linear iteration. Furthermore, more effective pre-conditioning



	32-cores		256-cores	
KVLCC2-15.8m	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	2.72E-01	-	3.08E-02	-
P-GMRES	4.06E-01	1.49	4.47E-02	1.45
Chaotic	1.87E+00	6.88	1.48E-01	4.81
KVLCC2-10.0m	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	1.48E-01	-	1.79E-02	-
P-GMRES	2.37E-01	1.61	2.65E-02	1.48
Chaotic	6.07E-01	4.11	7.05E-02	3.94
KVLCC2-5.99m	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	8.04E-02	-	8.63E-03	-
P-GMRES	1.23E-01	1.53	1.15E-02	1.34
Chaotic	2.83E-01	3.52	2.33E-02	2.70
KVLCC2-2.67m	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	2.98E-02	-	3.19E-03	-
P-GMRES	4.69E-02	1.58	4.29E-03	1.35
Chaotic	9.00E-02	3.02	6.65E-03	2.09
KVLCC2-1.11m	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	8.39E-03	-	2.09E-03	-
P-GMRES	1.20E-02	1.43	2.09E-03	1.00
Chaotic	2.03E-02	2.42	1.93E-03	0.93
LDCF-1.00m	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	3.95E-03	-	2.63E-03	-
P-GMRES	5.57E-03	1.41	2.67E-03	1.01
Chaotic	1.05E-02	2.65	1.59E-03	0.61
KVLCC2-317k	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	1.57E-03	-	1.37E-03	-
P-GMRES	1.91E-03	1.21	1.73E-03	1.27
Chaotic	2.08E-03	1.32	4.41E-04	0.32
NACA0015C-28k	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	9.68E-04	-	1.60E-03	-
P-GMRES	8.30E-04	0.86	1.47E-03	0.92
Chaotic	3.22E-04	0.33	1.79E-04	0.11
Dambreak-16k	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	6.24E-04	-	1.54E-03	-
P-GMRES	8.86E-04	1.42	1.49E-03	0.97
Chaotic	3.13E-04	0.50	3.01E-04	0.20
LDCF-14.4k	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	7.69E-04	-	1.42E-03	-
P-GMRES	7.48E-04	0.97	1.86E-03	1.31
Chaotic	1.99E-04	0.26	5.03E-04	0.35
Cylinder-4.3k	**Time (s)**	**Factor**	**Time**	**Factor**
GMRES	8.06E-04	-	1.68E-03	-
P-GMRES	8.78E-04	1.09	1.66E-03	0.99
Chaotic	1.96E-04	0.24	3.64E-04	0.22

Figure 4: Performance results for the three solvers (GMRES, PGMRES and Chaotic). Scalability graphs show speed-up relative to 32 cores. The tabular results show absolute wall-time and how this benchmarks against GMRES (lower factor implies faster than GMRES). Results are based on average solve times for all equations (momentum, pressure and additional transport equations over 50 non-linear iterations).

of the KSP methods may make them more favourable – assuming scalability can be maintained.

For other applications, where the convergence of the linear system is important, Chaotic solvers could be used in conjunction with more complex methods – either as a preconditioner or a hybrid solver. It is very simple to detect when the Chaotic methods begin to stagnate (corresponding to the point where high-frequency errors have been smoothed); and the solver could dynamically switch at this point. For CFD, this could also be viable, and further analysis will be performed along with improvements to the Chaotic solver itself.

The performance of the Chaotic solver could be improved further by modifying the stopping criteria evaluation, in order to prevent over-convergence of very small cases, and also to remove the blocking reduction used to compute the residual. One concept is to attach the local component of the residual to the SpMV communication, and have a delayed residual computed for free as these messages are passed through all the processes. Another option is to use non-blocking collectives from the MPI-3.0 standard to allow other communications to continue (however, early attempts at this were slow). Perhaps the best solution would be a hybrid, whereby the former option is used, but an early-stop routine could be built which forces an immediate residual-check when the local portion of the solution is well below the required convergence.

Ensuring the OpenMP threads stay 'alive' between the individual calls to the solver should be simple, and will reduce serial bottlenecks further.

Improving MPI communication routines for higher asynchronicity is likely to be a good area for improvement. The best way to achieve this would be through one-sided communication and hardware-level RDMA (Remote Direct Memory Access). This would be particularly hard to implement, especially without invoking high initialization costs - and requires significant changes to PETSc and ReFRESCO.

The implementation of the Chaotic solver, using a hybrid MPI + OpenMP parallelization scheme, is well-suited to acceleration using Graphics Processings Units (GPUs) or co-processors, and is also an area for investigation.

REFERENCES

[1] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F. & Zhang, H. (2013). PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory. `http://www.mcs.anl.gov/petsc`.

[2] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. & der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA.

[3] Chazan, D. & Miranker, W. (1969, April). Chaotic Relaxation. *Linear Algebra and its Applications*, 2(2):199–222.

[4] Davis, T. A. & Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25. ISSN 0098-3500. doi:10.1145/2049662.2049663.

[5] Ghysels, P., Ashby, T. J., Meerbergen, K. & Vanroose, W. (2013). Hiding global communication latency in the GMRES algorithm on massively parallel machines. *Journal of Scientific Computing*,

35(1):48–71.

[6] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B. & Vaz, G. (2014, October). Performance Analysis Of Massively-Parallel Computational Fluid Dynamics. The 11th International Conference on Hydrodynamics (ICHD), Singapore.

[7] Hawkes, J., Turnock, S. R., Cox, S. J., Phillips, A. B. & Vaz, G. (2014, September). Potential of Chaotic Iterative Solvers for CFD. The 17th Numerical Towing Tank Symposium (NuTTS 14), Marstrand, Sweden.

[8] Hoekstra, M. & Vaz, G. (2009, August). The Partial Cavity on a 2D Foil Revisited. In proceedings of the *7th International Symposium on Cavitation*, Ann Arbor, MI, USA.

[9] Horst, S. (2013, May). Why We Need Exascale And Why We Won't Get There By 2020. *Optical Interconnects Conference*, Santa Fe, New Mexico, USA.

[10] Klaij, C. & Vuik, C. (2013). Simple-Type Preconditioners for Cell-centered, Collocated, Finite Volume Discretization of Incompressible Reynolds-averaged Navier-Stokes Equations. *International Journal for Numerical Methods in Fluids*, 71(7):830–849.

[11] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, S. & Yelick, K. (2008, September). ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. *DARPA IPTO.*

[12] Menter, F., Kuntz, M. & Langtry, R. (2003, October). Ten Years of Industrial Experience with the SST Turbulence Model. In *Turbulence, Heat and Mass Transfer 4*. Antalya, Turkey.

[13] Rosetti, G., Vaz, G. & Fujarra, A. (2012, July). URANS Calculations for Smooth Circular Cylinder Flow in a Wide Range of Reynolds Numbers: Solution Verification and Validation. *Journal of Fluids Engineering, ASME*, page 549.

[14] Shalf, J. (2013, October). The Evolution of Programming Models in Response to Energy Efficiency Constraints. *Oklahoma Supercomputing Symposium*, Norman, Oklahoma, USA.

[15] Top 500 List (Acc. 2013, February). `http://www.top500.org`.

[16] Vaz, G., Jaouen, F. & Hoekstra, M. (2009, May–June). Free-Surface Viscous Flow Computations: Validation of URANS Code FRESCO. *28th International Conference on Ocean, Offshore and Arctic Engineering (OMAE)*, Honolulu, Hawaii.

J Mar Sci Technol
DOI 10.1007/s00773-017-0457-7

ORIGINAL ARTICLE

On the strong scalability of maritime CFD

J. Hawkes[1,2] · G. Vaz[2] · A. B. Phillips[3] · S. J. Cox[1] · S. R. Turnock[1]

Received: 26 September 2016 / Accepted: 16 May 2017

Abstract Since 2004, supercomputer growth has been constrained by energy efficiency rather than raw hardware speeds. To maintain exponential growth of overall computing power, a massive growth in parallelization is under way. To keep up with these changes, computational fluid dynamics (CFD) must improve its strong scalability—its ability to handle lower cells-per-core ratios and achieve finer-grain parallelization. A maritime-focused, unstructured, finite-volume code (ReFRESCO) is used to investigate the scalability problems for incompressible, viscous CFD using two classical test-cases. Existing research suggests that the linear equation-system solver is the main bottleneck to incompressible codes, due to the stiff Poisson pressure equation. Here, these results are expanded by analysing the reasons for this poor scalability. In particular, a number of alternative linear solvers and preconditioners are tested to determine if the scalability problem can be circumvented, including GMRES, Pipelined-GMRES, Flexible-GMRES and BCGS. Conventional block-wise preconditioners are tested, along with multi-grid preconditioners and smoothers in various configurations. Memory-bandwidth constraints and global communication patterns are found to be the main bottleneck, and no state-of-the-art solution techniques which solve the strong-scalability problem satisfactorily could be found. There is significant incentive for more research and development in this area.

Keywords High-performance computing · Strong scalability · Software profiling · Linear solvers

✉ J. Hawkes
j.hawkes@soton.ac.uk

1 University of Southampton, Boldrewood Campus, Southampton, United Kingdom

2 Maritiem Research Instituut Nederlands (MARIN), Wageningen, Netherlands

3 National Oceanography Centre, Southampton, United Kingdom

1 Introduction

A recent report by Slotnick et al. [28] attempted to create a "vision" of CFD in 2030, identifying some of the areas which must be improved to allow more widespread and successful use of CFD. Among these were improved turbulence and separation modelling; better automatic mesh generation and adaptivity; more capable multi-disciplinary simulations (for example, coupled CFD and structural simulations); improved post-processing, particularly of large simulations; greater accuracy through higher-order methods; and more practical design optimization. All of these goals require improvements to the underlying CFD algorithms—making them more efficient and more scalable—particularly considering the major changes in supercomputer architecture expected in the same era. Indeed, more scalable numerical methods are one of the areas highlighted by [28], stating that development has been stagnant for too long. The particular numerical methods that require improvement are not clear, and depend upon the type of CFD code and application.

Here, the strong scalability of CFD is reviewed in detail, using ReFRESCO. ReFRESCO is an incompressible-flow, unstructured, finite-volume, SIMPLE-based, segregated solver specialized for maritime applications; similar in formulation to many open-source and commercial codes. A general scalability study of the whole code has been performed (Sect. 4), which shows that the linear equation-system solver is the bottleneck in incompressible flow simulations. This study details the reason for their poor scalability, and shows that there are new problems facing scalable CFD since earlier literature [11], due to increasing memory bandwidth issues. Two test cases are used: lid-driven cavity flow (LDCF) and the KRISO Very Large Crude Carrier (KVLCC2) [20], each with around 2.67-million cells—chosen to show the full range of intra-nodal and inter-nodal scalability bottlenecks on the University of Southampton supercomputer (Iridis4).

Following this, a variety of other linear solvers and preconditioners are tested to determine whether the scalability problems can be circumvented (Sect. 5) and provide a comprehensive overview of the current 'best' solvers for strong scalability. These studies will aid CFD practioners in choosing suitable solvers and guide developers to find more scalable solutions. Widely-used solvers such as GMRES, Flexible-GMRES, BCGS and SOR are tested, along with state-of-the-art solvers such as Pipelined GMRES [10] which is designed to improve scalability. Similarly, multi-grid preconditioners such as Sandia's ML [9] could bring a significant improvement when compared to block-wise preconditioners (such as Block Jacobi) or simple smoothers (such as SOR).

2 The strong scalability problem

Over the last few decades, growth in supercomputing power has been exponential, with floating-point-operation (FLOP) rates doubling approximately every 14 months [29]. Whilst this growth is relatively constant, the underlying architectures which achieve such growth are not. Until 2004, the speed and electricity consumption of transistors was governed by Dennard's Scaling: as transistors shrank in size, their speed increased linearly and their electricity consumption dropped quadratically [7]. Unfortunately, the transistors used in modern processors are so small that electrons are able to 'leak' across the dielectric gates, and voltages must be increased to maintain stability. This limitation, known as the 'power wall', makes it more efficient for manufacturer's to provide multiple, slower cores in place of fewer, faster cores. Figure 1 shows the exponential growth rate of computing power and Fig. 2 shows how this computing power is provided—in terms of FLOP-rate-per-core and number of cores. The critical point in 2004 is clearly visible, where the shift towards a multi-core ('Chip-Level Multiprocessing') architecture occurred. This trend is relentless, and has lead to supercomputers with almost 200 cores-per-node. By 2020, it is expected that the cores-per-node ratio could be as high as 10-thousand; with the fastest supercomputers containing a total of 10- to 100-billion cores. Meanwhile, memory capacity is

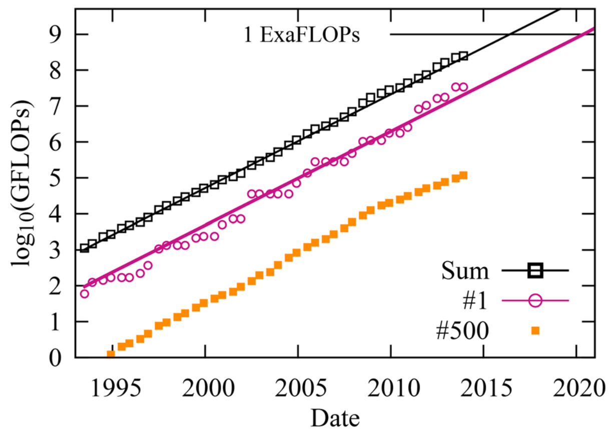


Fig. 1 Total floating-point operation (FLOP) rate of the most powerful 500 supercomputers in the world, using data from the Top500 organization [29]. The data shows the benchmark FLOP rate of the #1 machine, the #500 machine and the sum of all 500 machines over time. The exponential growth rate corresponds to a doubling every 13.85 months

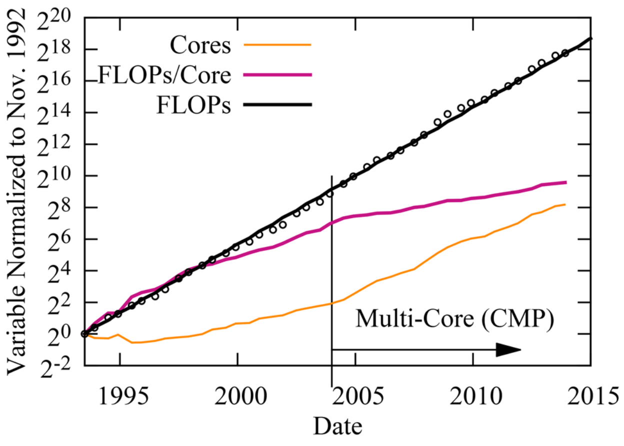


Fig. 2 The FLOP rate, FLOP/core ratio and number of cores (average of the Top500 [29]) over time, normalized to a snapshot in November 1993. Since the advent of chip-level-multiprocessing (CMP) in 2004, FLOP/core ratio has grown by only a factor of approx. 4, whereas number of cores has grown by a factor of approx. 64

growing at an ever-slower rate, limiting the absolute size of simulations; and memory bandwidth (per core) is decreasing, creating new issues for CFD algorithms [16, 19, 27].

It is often regarded that the CFD algorithm benefits from good 'weak scalability' (the ability to maintain computational efficiency with a fixed cells-per-core ratio), thus realizing the benefits of supercomputing advances when the growth in core-count was moderate [3]. Although difficult to quantify, it can be assumed that the capabilities of CFD have grown more-or-less in proportion with supercomputing power because of this trait; especially given that the maximum problem size (limited by total memory) has more-or-less grown with the number of nodes.

The ‘strong scalability’ of CFD—the ability to decrease the cells-per-core ratio efficiently—is more important in a massively-parallel era; and is generally poor [3, 6]. By 2020, supercomputers are expected to contain approximately 3000-times more cores, but the size of CFD simulations can only increase by a factor of ≈ 11, thus the efficient cells-per-core ratio of CFD must drop by a factor of at least 250.[1] Furthermore, in the maritime industry the majority of state-of-the-art simulations are unsteady computations. Since it is difficult to parallelize the time domain, greater spatial domain-decomposition is required to improve CFD capabilities, requiring further improvements to strong scalability.

Whilst both forms of scalability have been investigated for incompressible CFD, the results are often subjective to the particular code and hardware, and difficult to generalize. In [3] the linear-equation system solver, particularly for the Poisson pressure equation, is the main bottleneck to scalability. Culpo [6] reinforces this by considering the main elements of the linear-equation system solver and how they could be improved. However, neither paper investigates the reason for their poor scalability in detail, and neither considers alternative solver algorithms. Groppet al. [11] provides a good grounding in both these areas, but is now 15 years old—and the conclusions drawn could be somewhat outdated due to changes in hardware.

3 Experimental setup using ReFRESCO

In order to conduct scalability experiments, a sample CFD code (ReFRESCO) is used on two classical test cases. In order to get information on the run-time of ReFRESCO, the code is profiled by injecting timers into the code in key places. The test cases are run across a range of core-counts, from 1 to 512, on the University of Southampton supercomputer: Iridis4. The various aspects of the experimental setup are discussed below.

3.1 ReFRESCO

ReFRESCO[2] is a viscous-flow CFD code that solves multiphase, unsteady, incompressible flows for unstructured meshes [30]. It is complemented by various turbulence, cavitation and volume-fraction models. In many ways, ReFRESCO represents a general-purpose CFD code, with state-of-the-art features such as moving, sliding and deforming grids and automatic grid refinement—but it has been verified, validated and optimized for numerous maritime industry problems. ReFRESCO is currently being developed at MARIN (Netherlands) and a number of universities around the world [2, 8, 18, 24, 26] including the University of Southampton [12–14].

In ReFRESCO, the governing equations are discretized in strong-conservation form using a finite-volume approach with cell-centred collocated variables. Simulations are parallelized with MPI (Message Passing Interface) and partitioned using METIS [17]. ReFRESCO is based on the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) solver with pressure-weighted interpolation (PWI) [18].

The SIMPLE algorithm is shown in Fig. 3. The coarsest loop in the SIMPLE algorithm is responsible for unsteady time-stepping. Time integration is performed implicitly with first- or second-order backward schemes. All non-linearity is tackled in the ‘outer loop’, which is performed several times per time-step (until satisfactory convergence).

In each outer loop, a Picard-linearized version of each transport equation is assembled into a system of linear equations, based on the discretization of all the equation terms (time-derivatives, convection, diffusion, source terms) in the associated mesh. This process creates a sparse matrix of implicit terms ($\mathbf{A}$) and a vector of explicit terms ($\mathbf{b}$); which can be solved to find new values for the flow field ($\mathbf{x}$): $\mathbf{Ax} = \mathbf{b}$. Iterative solvers are used to solve for $\mathbf{x}$ to a given convergence tolerance (based on the ℓ^2-norm of the residual) in an ‘inner loop’, with typical tolerances between 0.1 and 0.001. ReFRESCO uses PETSc (Portable Extensible Toolkit for Scientific Computing) [1] for its large range of linear solvers and preconditioners.

Since each MPI process has its own memory space and its own portion of the mesh, the updated values for $\mathbf{x}$ along partition boundaries must be shared via MPI data exchange. The gradient of the flow-field variable can then be computed using Gauss theorem, and the updated gradients exchanged across domain boundaries once again.

These four routines {assembly, solve, exchange and gradients}, which can be seen in Fig. 3, are the heart of the SIMPLE algorithm and are the most expensive part of the solution process. They are also the linchpin of other SIMPLE-derived algorithms and common alternatives such as SIMPLEC, SIMPLER and PISO.

In order to observe how these key routines scale, ReFRESCO has been profiled using Score-P [VI-HPS Acc. 2013]. Score-P is a compile-time wrapper which automatically logs various run-time events, such as function

[1] Based on Tianhe-2, the current (January 2016) #1 supercomputer and conservative estimates of an exascale machine expected in 2020. Maximum capabilities of CFD are based on trends of total computational power, doubling every 13.85 months. Realistically, maximum CFD capabilities are limited by memory capacity which grows even more slowly [19], thus amplifying the problem.

[2] See http://www.refresco.org.

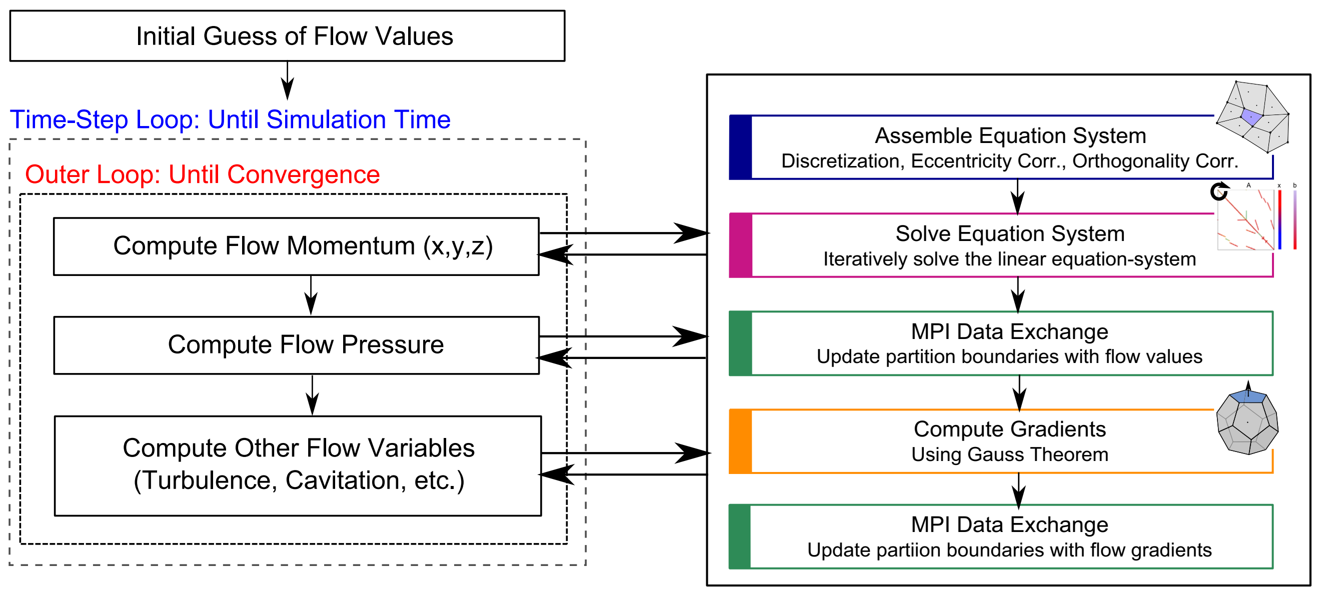

Initial Guess of Flow Values
Time-Step Loop: Until Simulation Time
Outer Loop: Until Convergence
Compute Flow Momentum (x,y,z)
Compute Flow Pressure
Compute Other Flow Variables (Turbulence, Cavitation, etc.)
Assemble Equation System
Discretization, Eccentricity Corr., Orthogonality Corr.
Solve Equation System
Iteratively solve the linear equation-system
MPI Data Exchange
Update partition boundaries with flow values
Compute Gradients
Using Gauss Theorem
MPI Data Exchange
Update partiion boundaries with flow gradients

Fig. 3 An overview of the SIMPLE algorithm. Time discretization is handled by the time-step loop at the coarsest level. Within each time-step a number of outer loops are performed in order to solve non-linearity and couple the governing equations. Within each outer loop, the solutions to the governing equations (momentum, pressure, etc.) are computed in their uncoupled, linearized, discretized form. The solution of each equation follows the same five steps as illustrated on the *right*

calls and durations. It has been carefully filtered such that only critical functions are measured, and the effect on total run-time is less than 2%. Score-P can be linked to PAPI (Performance Application Programming Interface) [4] which gathers additional information from physical hardware-counters.

Two hardware counters are enabled for these tests. The first measures floating point operations per second, which helps to determine whether the processing units are saturated. For an unstructured CFD code this is an unlikely situation, as the indirect memory access bottlenecks the processor. Structured-mesh codes are more likely to reach these limits due to better memory layout and vectorization.

Hardware counters for pure memory bandwidth are not available, but level one (L1) cache misses give a good indication of memory-fetching issues (since a cache miss must result in a memory or next-level cache transaction). However, there are two issues. Firstly, the compiler will often prefetch memory into the cache, resulting in memory bandwidth usage which is not detected by this hardware counter. Secondly, an L1 cache-miss will be registered even when the data resides in L2 or shared cache (which is still very fast compared to off-chip memory). Despite these imperfections, it is possible to see certain bottlenecks due to memory bandwidth, particularly when combined with other hardware counters or profiling.

Other hardware counters are available (such as L2 cache misses), but they cannot be enabled at the same time due to register sizes or competing circuitry.

3.2 Iridis4

ReFRESCO is run on the University of Southampton's latest supercomputer. Iridis4 has 750 nodes, consisting of two Intel Xeon E5-2670 Sandybridge processors (8 cores, 2.6 Ghz), for a total of 12,200 cores. Each 16-core node is diskless, but is connected to a parallel file system, and has 64GB of memory. The nodes run Red Hat Enterprise Linux version 6.3. Nodes are grouped into sets of 30, which communicate via 14 Gbit/s Infiniband. Each of these groups is connected to a leaf switch, and inter-switch communication is then via four 10 Gbit/s Infiniband connections to each of the core switches. Management functions are controlled via an ethernet network.

Iridis4 ranked #179 on the Top500 list of November 2013 with a peak performance of 227 TFLOPS [29]. Iridis4 cannot be classified as a next-generation, many-core machine, with only 16 cores per node. Indeed, it is several years behind the state-of-the-art. Nonetheless, it should be able to give sensible insight into the limitations of the CFD algorithm.

3.3 LDCF and KVLCC2

Two test cases are used in the experiments. The first is a laminar-flow, canonical, unit-length, three-dimensional lid-driven cavity flow (LDCF). A uniform structured mesh of 2.68-million cells (139^3) is used, and only momentum and pressure equations are solved. The simulation mimics an infinite domain, with two cyclic boundary conditions. The remaining four boundaries are constrained with Dirichlet boundary conditions, one of which specifies a tangential, non-dimensional velocity of 1.

The second test case is the KRISO Very Large Crude Carrier (KVLCC2) double-body wind-tunnel model [20]. The mesh is a three-dimensional multi-block structured mesh consisting of 2.67-million cells. A k-ω, two-equation shear stress transport turbulence model is used [22]. The domain and mesh are shown in Figs. 4 and 5 respectively.

The two test-cases are designed to have a similar number of cells. The size of the mesh has been chosen to show the full range of scalability issues. On 512 cores, the cells-per-core ratio reaches approximiately 5200 which helps demonstrate the parallel bottlenecks on a large number of cores; yet the problem is substantial enough to show memory bandwidth issues on a single node (with approximately 167k cells-per-core).

In both cases, 400 outer-loops are performed with no time-stepping, with an inner-loop (linear) relative convergence tolerance of 0.1 (0.01 and 0.001 later). Relaxation is applied to all outer-loops to stabilize the non-linear iterative process. QUICK (Quadratic Upstream Interpolation for Convective Kinematics) and first-order upwind schemes are used to discretize the convective terms of the momentum and turbulence equations respectively. A GMRES (Generalized Minimal Residual method) solver is used with a Block Jacobi preconditioner, as in [11].

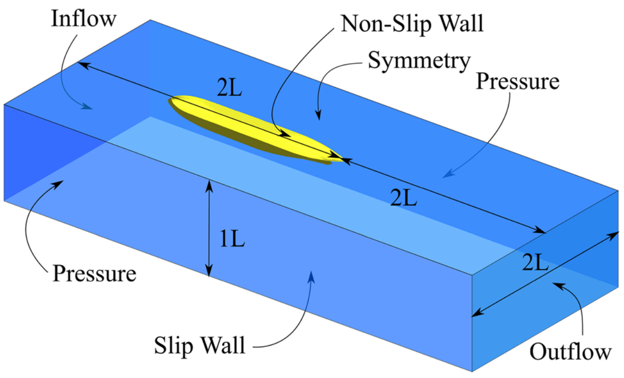


Fig. 4 The domain used for the KVLCC2 double-body wind-tunnel simulation. Symmetry boundary conditions are applied at the waterplane, but port- and starboard-sides of the centre-line are both simulated

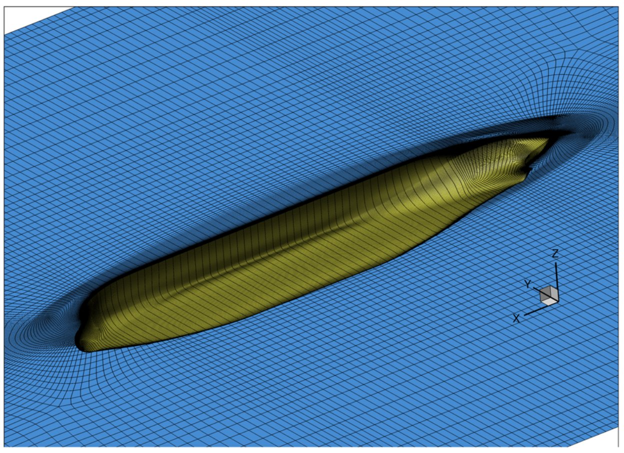


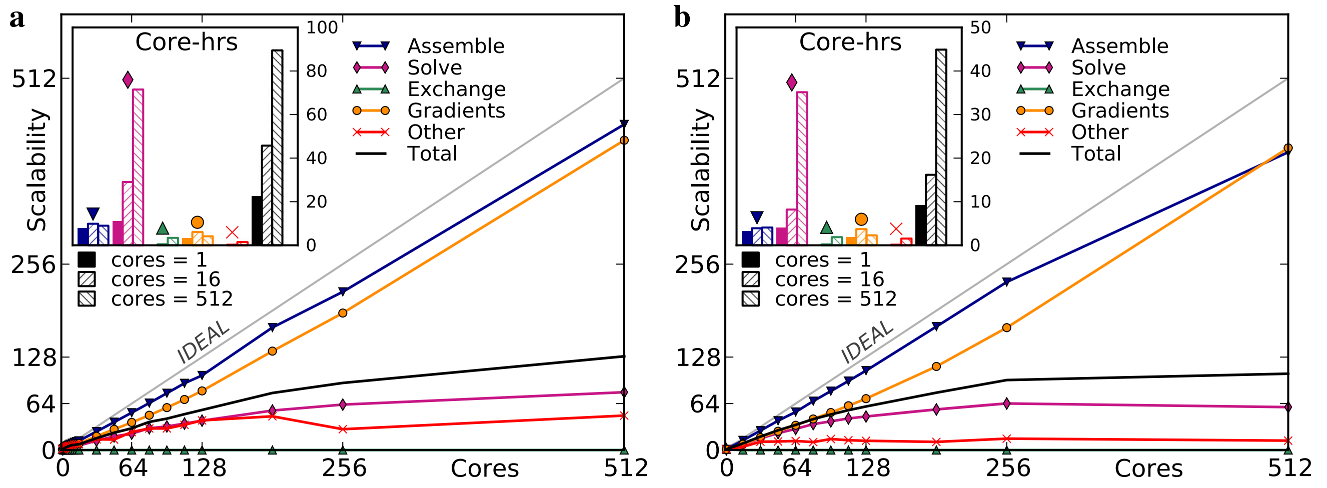
Fig. 5 The KVLCC2 mesh is a multi-block structured mesh consisting of 2.67-million cells

4 General scalability study

The scalability of the SIMPLE algorithm has been examined by measuring the run-time of the two test cases as successively more cores are added to the simulation. A scalability factor S can be defined as $S = T_1/T_C$ where T_C is the wall-time using C cores. Ideal scalability is when $S = C$, although this is rarely achieved. This scalability factor can be found for various subroutines in the SIMPLE algorithm in order to identify potential bottlenecks. Here, the scalability factor is normalized to serial operation ($C = 1$), and is often called a parallel 'speed-up' factor for this reason.

Figure 6a, b show the scalability of the code as the number of cores increases. The embedded bar charts show absolute core-hours ($C \times T_C$, which would ideally remain constant) in serial operation ($C = 1$), single-node operation ($C = 16$) and highly-parallel operation ($C = 512$). Total core-hours is shown with black bars; the composite parts are coloured and keyed with respect to the enclosing scalability plot. This shows the denormalized costs of various routines; and also highlights where the scalability bottlenecks occur—at the intra-nodal or inter-nodal level. The results from the hardware counters are shown in Fig. 7, for the LDCF test case.

Total scalability is poor overall, suffering significantly on just 16 cores due to intra-nodal bottlenecks and worsening to the point at which almost no speed-up is gained from adding additional nodes. On 512 cores the parallel speed-up is just 128 (KVLCC2) and 100 (LDCF). This scalability is similar to other codes, such as OpenFOAM [6, 25], STAR-CCM+ [5] or Ansys Fluent [15] although some published results are normalized to nodal performance (i.e. $C = 16$, which hides intra-nodal inefficiency and gives overly-optimistic results) or are truncated (hiding inter-nodal bottlenecks). Exact comparisons between various codes on identical hardware were not feasible, but only minor differences are expected since all of these packages are based off the same algorithm and share similar implementations. Major differences should only be observed if coupled solvers or inferior partitioning schemes are used.



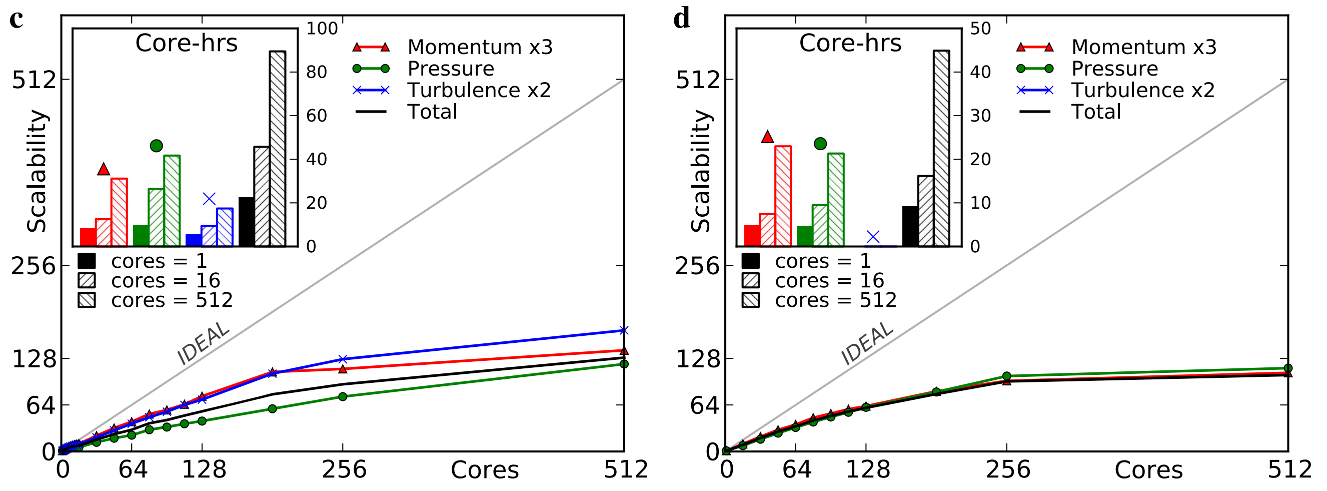


Fig. 6 Scalability of the code, and the profiled routines within, as the number of cores increases. These results used GMRES as the linear solver with a Block Jacobi preconditioner, and an inner-loop relative convergence tolerance of 0.1. **a** KVLCC2 breakdown by routine, **b** LDCF breakdown by routine, **c** KVLCC2 breakdown by equation, **d** LDCF breakdown by equation

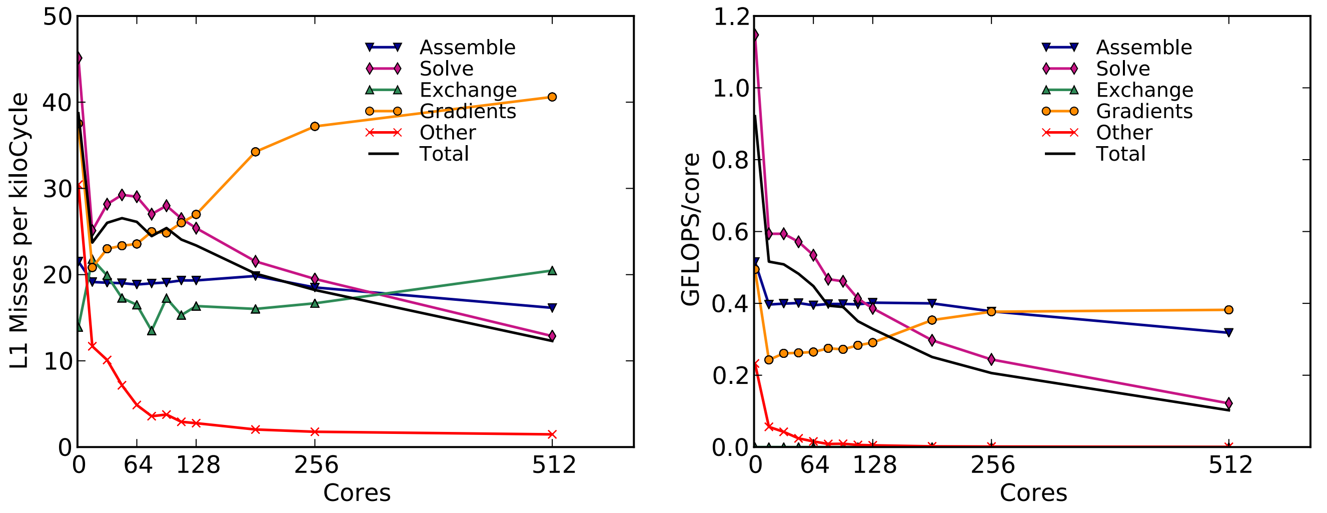


Fig. 7 An example of the information gleaned from PAPI hardware counters. For all routines in the LDCF test case, the number of first-level (L1) cache-misses per thousand clock cycles (*left*) and the total floating-point operation rate (FLOPs) (*right*) are shown

The routines outlined earlier assembly, solve, exchange and gradients, for each equation in each outer loop, account for the large majority of overall run-time. The remaining time (other) is spent (mostly) in one-off functions such as file IO or MPI initialization, thus is subjective to the length of the simulation and amount of IO required. These other routines may also increase significantly if additional features such as moving, deforming and adapative grids are used—again, this is highly subjective.

The assembly and gradients routines scale favourably, reaching almost 90% parallel efficiency. The high, and increasing, cache-miss rate of the gradients routines is curious, as it does not seem to affect performance (FLOP rate is maintained). It is likely that the data required resides in L2 or shared-cache, rather than off-chip memory, so the impact of these L1 cache misses is much lower.

The data exchange routines are not visible in the scalability plots, because normalizing against ($T_1 \approx 0$) gives negative scalability (no communications are required in serial operation). In reality, these routines scale reasonably—as the number of cores increases the size of the messages become smaller, and these messages can be sent concurrently. With inadequate load-balancing these data exchanges can become costly, due to the implicit synchronization of MPI processes. However, the results show that these communications account for a small proportion of overall run-time.

As consistent with literature, the solve routines have poor scaling and are a major contributor to total run-time, thus are the main concern for scalability. The hardware performance counters show a high cache-miss rate between 16 and 128 cores, corresponding to saturated memory bandwidth at the intra-nodal level.

Memory-bandwidth-per-node has been growing at approximately half the rate of processing-power-per-node leading to today's problems with memory bandwidth [27]. In [11], conducted in 2000 on single-core processors, memory-bandwidth problems were apparent but not as concerning—changes in architecture over the last decade have made memory bandwidth issues more critical.

Beyond 128 cores, cache misses in the solve routines become less frequent but FLOP rate continues to decrease and scalability worsens. This is due to the oft-observed global communication bottleneck. An illustration of a single GMRES iteration is shown in Fig. 8. This pattern is computationally similar to most Krylov Subspace (KSP) solvers, including conjugate gradient methods—although some differences will be mentioned in Sect. 5. In particular, two distinct communication routines are required by KSP solvers.

Firstly, sparse-matrix-vector-multiplication (SpMV) requires concurrent neighbour-to-neighbour communication as in the data exchange routines—scaling reasonably well.

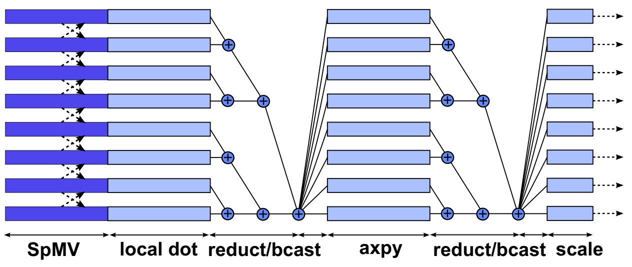


Fig. 8 An illustrative trace of a GMRES iteration on 8 cores (not to scale). Note the local neighbour-to-neighbour communications performed asynchronously in the SpMV routine, and the two reduction-broadcast patterns. There are many variations of the reduction-broadcast algorithm which cannot be illustrated clearly. The most common is the 'butterfly' algorithm which completes in $\log_2(C)$-time, combining the reduction and broadcast into a single hierarchy of latency-bound messages. As the number of cores increases, this reduction-broadcast takes longer, whilst other routines take less time

Secondly, two global reduction-broadcast routines are required for orthogonalization and normalization of the Krylov vectors. These require a hierarchical global communication pattern which scales poorly, usually with $T_C \underset{\sim}{\propto} \log_2(C)$ (where the proportionality factor depends primarily on network latency). These communication patterns are blocking, and cannot easily be overlapped or hidden by other useful work. On a large number of cores spread over an inter-nodal network with relatively high latency, these global communications become a bottleneck to scalability of the linear solvers. Whilst most routines reduce in wall-time as more cores are added (with less-than-ideal efficiency), wall-time for global communications increases, as each time the number of cores doubles, an extra set of messages must be sent (incurring the latency cost of the network).

These global communications create a scalability bottleneck when a high number of nodes are used, and has been well-documented in the literature [6]. The memory-bandwidth problems previously noted are often overlooked, but are an important bottleneck to overcome for next-generation supercomputing.

Figure 6c, d shows the breakdown of time spent in the various equations (pressure, momentum, turbulence). In both cases, the single pressure equation took considerable time to compute—similar to all three momentum equations combined. In incompressible-flow simulations the pressure equation is much harder to solve than other transport equations, due to its elliptic Poisson form.

This can be illustrated by considering the spectral radius (maximum eigenvalue) of the Jacobi iteration matrix: $\rho(\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U}))$, where $\mathbf{D}$, $\mathbf{L}$ and $\mathbf{U}$ are the diagonal, lower and upper triangles of $\mathbf{A}$ respectively. The rate of convergence of the Jacobi method is proportional to the spectral radius, and must be less than unity for convergence. Using

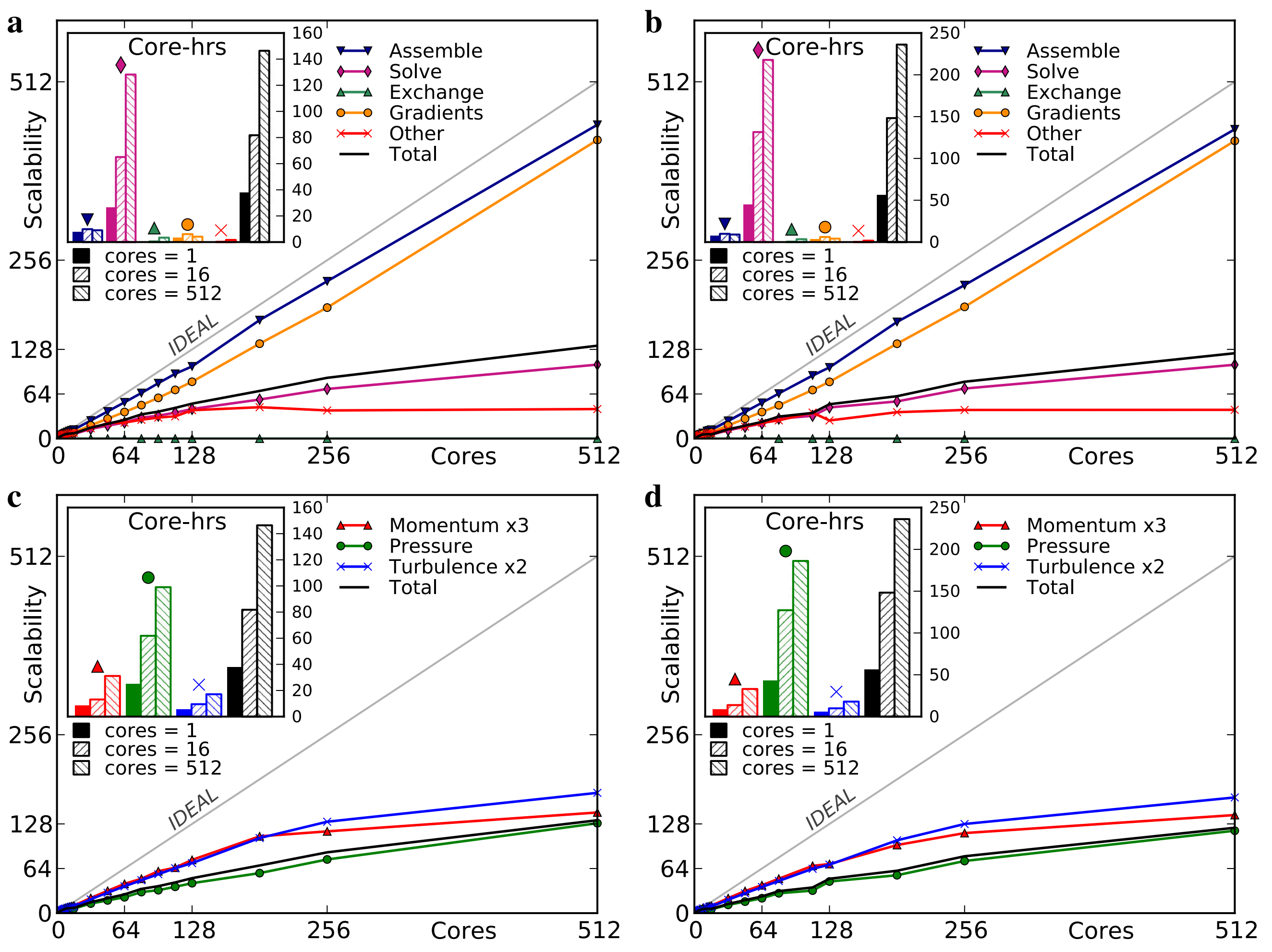


Fig. 9 Scalability of the code with an inner-loop convergence tolerance of **a**, **c** 0.01 and **b**, **d** 0.001 for the KVLCC2 test case as in Fig. 6, showing scalability of **a**, **b** the various routines and **c**, **d**) the various equations, using GMRES with a Block Jacobi preconditioner. The results show poor scaling of the solve routines, which particularly influences the cost of the pressure equation. Similar results were obtained for the LDCF test case

a smaller version of the KVLCC2 mesh (317k cells), the spectral radius of the KVLCC2 pressure equation was >0.9999, compared to 0.8450 for the momentum equation and 0.5888 for the turbulence equation.[3] Thus it would take at least 1700-times more iterations of the pressure equation to reach the same convergence as in the momentum equation, if using a naïve Jacobi solver. The pressure equation also gets stiffer as the mesh gets larger, amplifying the problem. KSP methods, particularly with good preconditioning, close this gap considerably, but there is still a large difference between the pressure equation and other transport Eq. (14).

The above results were performed using an inner-loop (linear) relative convergence tolerance of 0.1 for all equations. The results were repeated using a tolerance of 0.01 and 0.001 (see Fig. 9). Note the significant increase in wall-time, entirely due to the solve routines for the pressure equation.

These results were also re-run using alternative convective discretization schemes [12], which had no significant effect on scalability. Higher order methods, such as QUICK, apply their high-order terms explicitly (into the **b** vector); with only the low-order terms affecting the matrix (**A**). It is suggested that higher-order discretization will be a prominent development in the next decade [28], so this is a promising result. However, there are difficulties when going to even higher-order methods, and this remains an area of active linear-solver development [23].

A fully unstructured mesh was also tested using the KVLCC2 test case [12]. The unstructured mesh was much

[3] The spectral radii were found by extracting the matrices from the fifth outer loop in a separate batch of simulations. The maximum eigenvalue, and thus spectral radius, of the corresponding Jacobi iteration matrix could be found using ARPACK routines [21] based on Arnoldi iterative methods. The size of the matrix that could be tested was limited by memory capacity.

larger (12.5 m) and did not directly match a structured mesh, so the scalabiliy of two structured meshes was interpolated (10.0 and 15.8 m). The interpolated scalability of the structured meshes was virtually identical to the unstructured mesh. Note that ReFRESCO treats all meshes as unstructured meshes, in terms of data structure—and therefore does not take advantage of structured meshes and structured memory layout.

This preliminary study concludes that in their basic form, the linear equation-system solvers are the primary bottleneck to strong scalability of CFD, in agreement with recent literature. In particular, detailed profiling of ReFRESCO has revealed that memory bandwidth contention and expensive hierarchical communication patterns are the main bottleneck. However, the scalability may be significantly altered if different solvers and preconditioners are used.

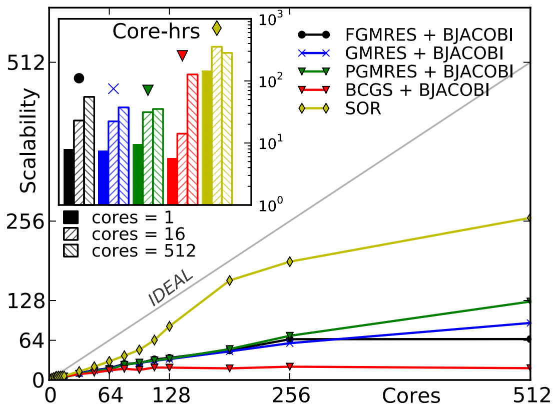


Fig. 10 Scalability of the solve routines in the pressure equation, using the KVLCC2 test case with inner-loop convergence tolerance set to 0.1. The results compare three Krylov Subspace solvers and a Successive Over-Relaxation (SOR) algorithm operating as a Block Gauss-Seidel method. Note the exponential scale of the inset core-hours chart

5 Effect of linear solvers and preconditioners on scalability

Thus far, the results have used a basic GMRES solver with a Block Jacobi preconditioner. More modern solvers or preconditioners could provide very different scalability characteristics. For example, a powerful preconditioner could reduce the number of KSP iterations (and global communications) required; but may be unscalable in itself due to communication or high setup costs. CFD is unique in that a solution to the linear system is only approximated, since the solution to the linear system is a small part of a non-linear system. Compared to applications which require machine accuracy of linear systems, CFD is very sensitive to start-up (initialization of linear solvers, memory allocation, etc.) costs which may rule out the most advanced solvers or preconditioners. Indeed, it may be that simpler preconditioners than Block Jacobi provide better scaling. In this section, the scalability of the linear solvers will be tested. Following this, a number of preconditioning techniques will be investigated—including block preconditioning techniques, multi-grid methods and simple smoothers.

5.1 Solvers

A recent improvement to GMRES has been developed, so-called Pipelined GMRES (PGMRES) [10], which removes one of the global communications from the standard GMRES iteration—replacing it with a correction routine, and allowing the remaining reduction to be overlapped with other useful work. The scalability of GMRES and PGMRES are compared in Fig. 10. Flexible GMRES (FGMRES) has been tested, as it allows a wider range of preconditioning techniques to be used later. A Bi-Conjugate Gradient Squared (BCGS) method has also been tested. All of the KSP methods use Block Jacobi as a preconditioner. A successive over-relaxation (SOR) method is also shown, demonstrating the differences between KSP and non-KSP methods.

GMRES performs as previously noted, with poor performance at the intra-nodal level due to memory-bandwidth contention and poor performance on a large number of cores due to global communication patterns. FGMRES exhibits slightly worse inter-nodal scalability than GMRES. PGMRES scales worse in the memory-bandwidth zone than either GMRES or FGMRES, but the gradient of the scalability factor, dS/dC, between 256 and 512 cores is approximately double that of GMRES—consistent with the algorithmic improvement (half the number of global communications). Overall, the wall-time gains from PGMRES on 512 cores are minimal.

BCGS gives strong numerical performance in serial operation and similar memory-limited scaling at the intra-nodal level. BCGS requires four global communications per iteration, thus inter-nodal scaling suffers.

As expected, SOR performs much worse than the KSP methods, but has far superior scalability. SOR is limited by memory bandwidth briefly, but quickly recovers this as the simulation fits into cache. The SOR algorithm still uses one global communication pattern to compute a residual at the end of each iteration; thus its final gradient is similar to that of PGMRES. Residuals could be calculated less frequently to improve scalability of SOR considerably.

ReFRESCO also has access to a large range of preconditioners through its use of PETSc, many of which have been tested as follows.

5.2 Block preconditioners

The Block Jacobi algorithm used thus far implements a block-wise Incomplete LU (ILU) factorization with zero-level fill, and sets a high benchmark for other preconditioners. ILU(0) is performed on each MPI process's local portion of the matrix, leading to an interesting problem: convergence deteriorates as the number of cores increases, as the local portion becomes less significant to the global solution.[4] An Additive Schwarz Method (ASM) was tested, with the same block-wise ILU(0) solver. ASM is similar to Block Jacobi, but allows communication between neighbouring blocks to augment the process. The results are shown in Fig. 11. The differences between Block Jacobi and ASM were small, with ASM fairing worse overall due to additional communications. ILU(0), ILU(1) and ILU(2) were also tested as preconditioners in their own right, with poor results in all regards (not shown).

5.3 Multi-grid preconditioners

For elliptic equations (such as the pressure equation) multi-grid methods such as ML [9] should be very powerful. Multi-grid methods cover a broad category, with multiple formulations and many opportunities for fine-tuning. They are all based on the principle that multiple scales of the problem can be solved efficiently by solving coarse-grid approximations to the actual (fine) grid. The coarsest grid will have a much lower spectral radius than the finest, allowing low-frequency errors to be reduced quickly. Meanwhile, the fine grid solves high-frequency errors, and the results are combined. Since each grid is much easier to solve, 'smoothers' are used instead of complete solvers at each level. A typical smoother may just be one iteration of SOR or an ILU factorization, for example, although the coarsest grid is often solved directly. There are many variations of multigrid methods: different methods for coarse-grid construction; different methods for coarse-grid interpolation; various methods of communicating (or not) on coarse grids; and so on. All of these variations will have a large effect on scalability.

ML is a state-of-the-art smoothed-aggregation algebraic multi-grid method from Sandia's National Labaratories, and is one of the most commonly-used multigrid packages [9]. ML automatically creates coarse grids until a minimum size is reached using a smoothed aggregation process. For the KVLCC2 test case, ML automatically decided to create

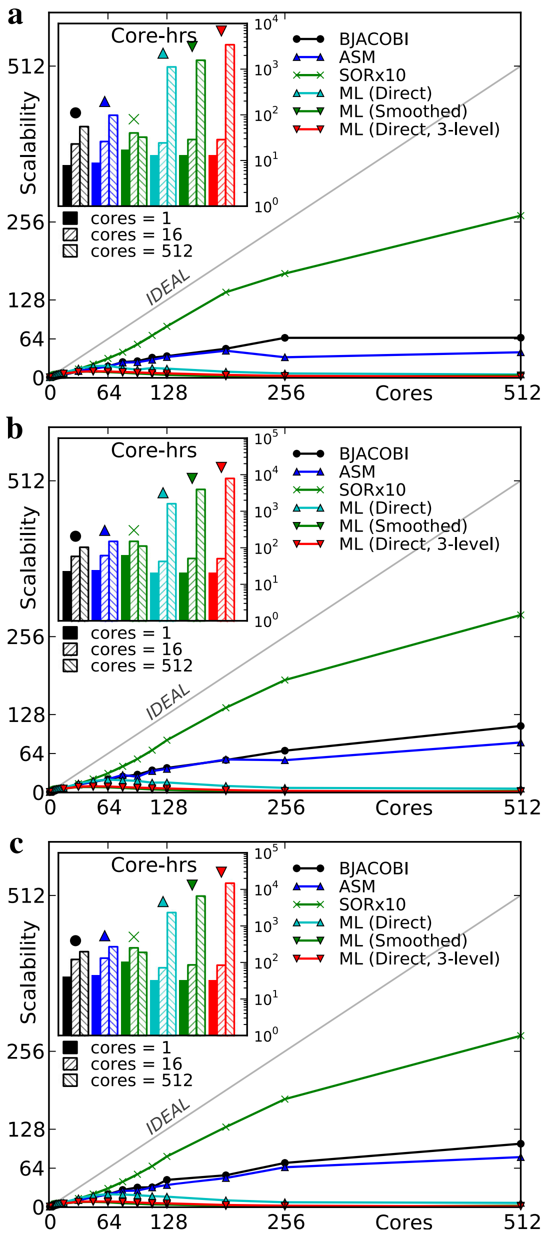


Fig. 11 Scalability of the solve routines in the pressure equation, using the KVLCC2 test case with inner-loop convergence tolerance set to **a** 0.1, **b** 0.01 and **c** 0.001. The results compare different preconditioners including Block Jacobi, Additive Schwarz (ASM), SOR smoothing (SORx10) and a multi-grid method (ML). FGMRES is used as the solver to allow more flexible preconditioning. Note the exponential scale of the inset core-hours chart

six grids when running on one core, and four grids on 512 cores. FGMRES was necessary to accommodate the multi-grid preconditioner, because the preconditioning matrix could change between iterations.

The results shown in Fig. 11 show that ML is highly capable at the intra-nodal level. It exhibits strong serial performance and moderate scaling to 16 cores—better than Block Jacobi. Unfortunately it rapidly breaks down beyond

[4] Indeed, this made it difficult to distinguish between convergence-loss-problems and global-communication-problems in the previous section. Profiling of an unpreconditioned GMRES reveals that globals communications are the leading problem. However, the number of iterations required with Block Jacobi increased when a tolerance of 0.001 was requested.

64 cores where global communications dominate. It was suspected that this was due to the direct solver used on the coarsest grid, but replacing it with a smoother (5 iterations of SOR) resulted in worse scalability. Another recommendation is to restrict the number of levels created by ML, thus reducing expensive start-up costs. Restricting ML to three levels worsened the scalability; and two levels (not shown) gave similar results. It is expected that less sophisticated multigrid methods (such as non-smoothed aggregation) may provide better results for CFD, since start-up is cheaper. It is also possible to re-use the coarse-grid mappings between non-linear iterations, which would improve the results shown here. Furthermore, the multigrid method could be used as a standalone solver rather than a preconditioner, omitting FGMRES entirely. Clearly a much deeper study of multi-grid methods is required as they certainly cannot be used as a black-box for scalable CFD.

5.4 Smoothing as a preconditioner

Finally, it is worth considering a much simpler preconditioner than even Block Jacobi. Instead of preconditioning in the classic sense, a smoother can be used before the main solver (FGMRES). Ten iterations of SOR as a smoother was optimal (compared to 1, 100 or 1000). Although it performed worse than Block Jacobi in serial operation, where Block Jacobi has good convergence, it was able to utilize the super-linear scalability as noted in Sect. 4 due to caching of memory, thus providing excellent scalability. Since a fixed number of smoother iterations were performed, residual computation in the SOR algorithm was unnecessary, improving scalability further. This combination has minimal setup costs, since SOR requires no additional memory or pre-computation, so could be a viable option for scalable CFD. However, the solver is still unavoidably limited by memory bandwidth contention, and global reductions from the overruling KSP solver. Furthermore, the numerical performance is not good, so it is only competitive on a large number of cores.

This section has looked at various linear solver and preconditioners. The most promising result was from a SOR smoother instead of a classical preconditioner. Although it was the slowest configuration for serial computation, superior scaling meant that it was often the best performer at $C = 512$. A multi-grid preconditioner was tested, which performed well on a low number of cores, but had poor scaling. A more detailed study of multi-grid preconditioning may yield better results. A more scalable version of GMRES was also tested (Pipelined GMRES) with mixed results—global communication problems were halved; but at the cost of more memory-bandwidth problems.

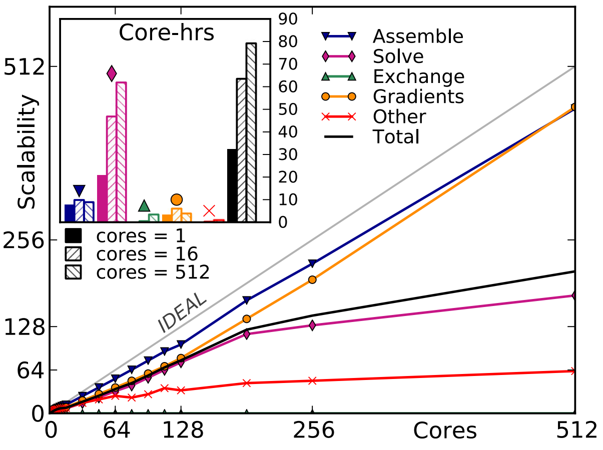


Fig. 12 Scalability of the code for the KVLCC2 test case, and the profiled routines within, as the number of cores increases. These results used FGMRES with 10 iterations of SOR as a smoother, with an inner-loop relative convergence tolerance of 0.1. Similar results were found for the LDCF test case

Overall, the best setup required using 10 iterations of SOR as a smoother/preconditioner. The initial study is re-illustrated using this configuration in Fig. 12. Inter-nodal scaling is significantly improved (compared to Fig. 6a), which is encouraging, but parallel efficiency is still poor and global communications are still limiting. Overall wall-time on 512 cores has been improved by approximately 30%, but with stricter convergence tolerances these gains are lost due to the poor numerical properties of SOR.

6 Conclusions

The main causes of inefficiency and poor scalability of the SIMPLE method have been analyzed by profiling the performance of a state-of-the-art CFD code from 1 to 512 cores. The results show that the main bottleneck is the linear equation-system solvers, particularly for the Poisson pressure equation. The main problem with the linear equation-system solvers is the large amount of expensive, unscalable global communications that are performed. Profiling with hardware counters has also revealed further problems at the intra-nodal level due to memory-bandwidth contention.

Experiments were performed in order to measure performance differences between various state-of-the-art linear equation-system solvers and preconditioners. Recent developments such as a 'pipelined' version of GMRES showed improved inter-nodal scalability but gave worse absolute speed and intra-nodal scalability—overall giving only a minor performance increase. Multigrid methods offer some hope, but their performance is nuanced and difficult to predict. The results showed that multigrid preconditioners were able to offer better absolute speed, but did not scale as well as simpler preconditioners such as Block Jacobi. Depending on the convergence tolerance of the linear equation system, simple smoothers often gave the best performance. Replacing the classical Block Jacobi preconditioner with ten iterations of Successive Overrelaxtion improved overall wall-time on 512 cores by 30%.

By 2020, supercomputers are expected to be 3000-times more parallel [19], with total power (and practical simulation size) growing by a factor of just $\approx$11. Based on these hardware predictions, the cells-per-core ratio must drop by a factor of at least 250 in order to maintain a practical simulation time. Today, a practical simulation of 50-million elements, using 512 cores of Iridis4, would achieve a cells-per-core ratio of approximately 100-thousand. Dividing this by 250 gives a predicted cells-per-core ratio of just 391. The results have shown that scalability at 5200 cells-per-core is already poor, with a maximum parallel efficiency of 25–50%. Beyond 512 cores, negative scalability is likely, with simulation time increasing as more cores are added. With the current state of the CFD algorithm, extrapolating to less than 500 cells-per-core is almost inconceivable.

There are currently developments to improve on the presented SOR results using 'chaotic' iterative methods, which provide even better scalability by removing the implicit synchronization in the sparse-matrix-vector communications; improving cache-use at the intra-nodal level; and slightly improving convergence rates. These chaotic methods could also be used to accelerate multigrid methods [13, 14]. Regardless of the specific methods, more research is needed to carry incompressible CFD codes into the next era of supercomputing, where many-core machines (including GPU or co-processor architectures) will be commonplace. Minor improvements can be expected from alternative software models (such as hybrid parallelization), but significant changes are required at the algorithmic level to keep up with rapidly-evolving hardware.

Acknowledgements Over 3000 simulations have been performed to obtain the results presented herein. The authors acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work. The authors would also like to thank C.M. Klaij (MARIN) for his guidance and expertise.

References

1. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Rupp K, Smith BF, Zhang H (2013) PETSc Users Manual. Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory, http://www.mcs.anl.gov/petsc
2. Bandringa H, Verstappen R, Wubbs F, Klaij C, Ploeg A (2012) On novel simulation methods for complex flows in maritime applications, Numerical Towing Tank Symposium (NUTTS). Cortona
3. Bhushan S, Carrica P, Yang J, Stern F (2011) Scalability studies and large grid computations for surface combatant using CFDShip-Iowa. IJHPCA 25(4):466–487
4. Browne S, Dongarra J, Garner N, Ho G, Mucci P (2000) A portable programming interface for performance evaluation on modern processors. Int J High Perform Comput Appl 14(3):189–204. doi:10.1177/109434200001400303
5. CD-Adapco (2010) Star-CCM+ Performance Benchmark and Profiling, HPC Advisory Council (Best Practices)
6. Culpo M (2011) Current bottlenecks in the scalability of openFOAM on massively parallel clusters. Tech. rep, Partnership for Advanced Computing in Europe
7. Dennard R, Gaensslen F, Rideout V, Bassous E, Leblanc A (1999) Design of ion-implanted MOSFET's with very small physical dimensions. Proc IEEE 87(4):668–678
8. Eca L, Hoekstra M (2012) Verification and validation for marine applications of CFD. In: 29th Symposium on Naval Hydrodynamics. Gothenburg
9. Gee M, Siefert C, Hu J, Tuminaro R, Sala M (2006) ML 5.0 Smoothed Aggregation User's Guide. Tech. Rep. SAND2006-2649, Sandia National Laboratories, Albuquerque
10. Ghysels P, Ashby TJ, Meerbergen K, Vanroose W (2013) Hiding global communication latency in the GMRES algorithm on massively parallel machines. J Sci Comput 35(1):48–71
11. Gropp W, Kaushik D, Keyes D, Smith B (2000) Analyzing the parallel scalability of an implicit unstructured mesh CFD code. In: Valero M, Prasanna V, Vajapeyam S (eds) High performance computing HiPC 2000. Lecture notes in computer science. Springer, Berlin Heidelberg, pp 395–404
12. Hawkes J, Turnock SR, Cox SJ, Phillips AB, Vaz G (2014a) Performance Analysis Of Massively-Parallel Computational Fluid Dynamics. In: The 11th International Conference on Hydrodynamics (ICHD), Singapore
13. Hawkes J, Turnock SR, Cox SJ, Phillips AB, Vaz G (2014b) Potential of Chaotic Iterative Solvers for CFD. In: The 17th Numerical Towing Tank Symposium (NuTTS 2014), Marstrand
14. Hawkes J, Turnock SR, Cox SJ, Phillips AB, Vaz G (2015) Chaotic Linear Equation-System Solvers for Unsteady CFD. In: The 6th International Conference on Computational Methods in Marine Engineering (MARINE 2015), Rome
15. Hewlett-Packard Development Company (2014) Scalability of ANSYS 15.0 Applications and Hardware Selection. Tech. rep
16. Horst S (2013) Why We Need Exascale And Why We Won't Get There By 2020. In: Optical Interconnects Conference, Santa Fe
17. Karypis G (2014) METIS: serial graph partitioning and fill-reducing matrix ordering, v5.1.0. Department of computer science and engineering, University of Minnesota, Minnesota. http://glaros.dtc.umn.edu/gkhome/fetch/papers/mlSIAMSC99.pdf
18. Klaij C, Vuik C (2013) Simple-type preconditioners for cell-centered, collocated, finite volume discretization of incompressible reynolds-averaged Navier-Stokes equations. Int J Numer Methods Fluids 71(7):830–849
19. Kogge P, Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J, Karp S,

Keckler S, Klein D, Lucas R, Richards M, Scarpelli A, Scott S, Snavely A, Sterling T, Williams S, Yelick K (2008) ExaScale computing study: technology challenges in achieving exascale systems, DARPA IPTO

20. Lee S, Kim H, Kim W, Van S (2003) Wind tunnel tests on flow characteristics of the KRISO 3,600 TEU container ship and 300K VLCC double-deck ship models. J Ship Res 47(1):24–38
21. Lehoucq RB, Sorensen DC, Yang C (1998) ARPACK Users' Guide: solution of large-scale eigenvalue problems by implicitely restarted Arnoldi methods. SIAM, Philadelphia
22. Menter F, Kuntz M, Langtry R (2003) Ten Years of Industrial Experience with the SST Turbulence Model. In: Turbulence, Heat and Mass Transfer 4, Antalya
23. Olson LN, Schroder JB (2011) Smoothed aggregation multigrid solvers for high-order discontinuous Galerkin methods for elliptic problems. J Comput Phys 230(18):6959–6976
24. Pereira F, Eca L, Vaz G (2013) On the order of grid convergence of the hybrid convection scheme for RANS codes. In: proceedings of CMNI, Bilbao, Spain
25. Pringle G (2010) Porting OpenFOAM to HECToR. The University of Edinburgh, EPCC
26. Rosetti G, Vaz G, Fujarra A (2012) URANS calculations for smooth circular cylinder flow in a wide range of Reynolds Numbers: solution verification and validation. J Fluids Eng 134(12):121103
27. Shalf J (2013) The evolution of programming models in response to energy efficiency constraints. In: Oklahoma Supercomputing Symposium, Norman
28. Slotnick J, Khodadoust A, Alonso J, Darmofal D, Gropp W, Lurie E, Mavriplis D (2014) CFD Vision 2030 Study: a path to revolutionary computational aerosciences. Tech. Rep. March, NASA Langley Research Center, Hampton http://www.ntrs.nasa.gov/search.jsp?R=20140003093
29. Top 500 List (Acc. 2013) http://www.top500.org
30. Vaz G, Jaouen F, Hoekstra M (2009) Free-surface viscous flow computations: validation of URANS code FRESCO. In: 28th International Conference on Ocean, Offshore and Arctic Engineering (OMAE), Honolulu
31. VI-HPS, Score-P, v123 (Acc. 2013) http://www.vi-hps.org/projects/score-p

Towards Exascale CFD: Chaotic Multigrid Methods for the Solution of Poisson Equations

J. Hawkes[a,b,*], G. Vaz[b], A.B. Phillips[c], C. Klaij[c], S.J. Cox[a], S.R. Turnock[a]

[a] *University of Southampton, Boldrewood Campus, Southampton, United Kingdom*
[b] *Maritiem Research Instituut Nederlands (MARIN), Wageningen, Netherlands*
[c] *National Oceanography Centre, Southampton, United Kingdom*

Abstract

Supercomputer power has been doubling approximately every 14 months for several decades, increasing the capabilties of scientific modelling at a similar rate. To utilize these machines effectively for CFD, improvements to strong scalability are required. The largest bottleneck to strong scalability is the parallel solution of linear Poisson equations. State of the art linear solvers such as Krylov Subspace or multigrid methods provide excellent numerical performance but do not scale efficiently, due to frequent synchronization between processes. Complete desynchronization is possible for simpler, Jacobi-like solvers using the theory of 'chaotic relaxations'. These non-deterministic, chaotic solvers scale superbly, as demonstrated herein, but lack the numerical performance to contribute seriously to exascale CFD – despite relatively lax convergence requirements. However, these chaotic princples can be translated to multigrid solvers. In this paper, a 'chaotic-cycle' algebraic multigrid method is described and implemented as an open-source library, and tested within the context of an industrial CFD code. The chaotic-cycle shows good scalability and numerical performance compared to classical V-, W- and F-cycle multigrid cycles. On 2048 cores the chaotic-cycle multigrid solver performs up to 7.7× faster than Flexible-GMRES and 13.3× faster than classical V-cycle multigrid; with room for improvement relating to coarse-grid communications and desynchronized residual computations.

Keywords: Exascale, Strong Scalability, Chaotic Methods, Multigrid, Chaotic Cycle, Chaos, CFD, ReFRESCO

1. Introduction

Up until approximately 2004, the speed and energy efficiency of supercomputers increased as transistors became smaller. However, below a critical transistor size, electrons begin 'leaking' across the dielectric gates, and voltages have to be increased to maintain stability. Manufacturers have resorted to packaging multiple cores onto a single chip to keep up with exponential growth expectations, and core clock-speeds have stagnated or decreased. By 2020, the first exascale computer, capable of 1 ExaFLOP (10^{18} floating-point operations per second), is expected. In order to achieve reasonable energy efficiency at such scale, the number of cores-per-node must continue increasing exponentially [1, 2, 3].

There are many proposed architectures for a many-core exascale machine, including accelerated machines (utilizing GPUs or Xeon Phi co-processors) or many-core CPU-based designs. The most powerful supercomputer according to the Top500 list [4] is *Sunway TaihuLight*, using 256-core CPUs to achieve a peak performance of 93 PFLOPS. In second place is *Tianhe-2*, which achieves 34 PFLOPS using Xeon Phi accelerators (192 cores-per-node). The current third ranked supercomputer uses NVidia k20 GPUs to

*Corresponding author
Email address: `j.hawkes@soton.ac.uk` (J. Hawkes)
URL: `https://bitbucket.org/jamesnhawkes/chaos` (J. Hawkes)

achieve 17.6 PFLOPS (2496 CUDA cores-per-node). An exascale machine would be 11-times more powerful than *Sunway TaihuLight*, but is estimated to be up to two orders-of-magnitude more parallel, with most of this parallelization coming at the intra-nodal (many-core) level.

In order for computational fluid dynamics (CFD) to keep up with this paradigm-shift, it is important to consider strong scalability – the ability to maintain parallel efficiency as the cells-per-core ratio decreases. Over the last decades, a modest increase in parallelization has not required significant changes to the fundamental algorithms driving CFD.

This paper concerns itself with semi-implicit methods for solving the incompressible Navier-Stokes equations on unstructured grids. Such CFD solvers are often based on the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm or derivatives such as SIMPLEC, SIMPLER or PISO. These methods linearize the governing equations for momentum, pressure, and other modelled quantities and iterate to find a solution to the non-linear problem, whilst coupling the equations at each iteration. In each SIMPLE iteration, a linear equation-system for each governing equation is assembled and must be solved to an acceptable tolerance, using a linear iterative solver.

It has been shown that repeatedly finding a solution to the linearized pressure equation is a severe bottleneck to the strong scalability of CFD [5]. The pressure equation is a stiff, elliptic, Poisson equation which contains many low-frequency error components [6]. Krylov Subspace (KSP) methods (such as residual minimization or conjugate gradient methods) are often employed to solve the pressure equation because they are numerically powerful and robustly converge low-freqency errors. Unfortunately, these methods require global communication patterns which scale very poorly. Efforts to hide or reduce the number of global communications (such as the 'pipelined' version of GMRES, P-GMRES [7], or improved bi-conjugate gradient methods such as IBCGS [8]) provide only minor improvements for CFD. The other equations, such as for momentum, are not elliptic and do not usually contain low-freqency errors. The high-frequency errors can be smoothed quickly using simple smoothers such as Jacobi or Gauss-Seidel methods which do not require global communications – only local neighbour-to-neighbour communications.

Although subjective to the details of the CFD solver and test case, the error in the linear equation-systems usually only needs to be reduced by one or two orders of magnitude – a lax relative convergence tolerance of 0.1 or 0.01. This is because the linear system is only an approximation of a larger non-linear system, and the repeated solution of the governing equations ensures overall convergence. It may be possible to use less powerful solvers on the pressure equation in order to achieve higher scalability. At any rate, it is unusual to require machine-accuracy of the linear solver, as is the case in other numerical fields – and this could be exploited.

Multigrid methods, such as ML [9], are gaining popularity due to their flexibility. By tuning the number of multigrid levels, the cycle type and number of pre- and post-smoothing steps it is possible to customize a solver which has a good compromise of numerical power (and therefore, absolute speed) and scalability. The main bottleneck to scalability of multigrid methods is the amount of implicit global synchronization, mostly triggered by the Jacobi or Gauss-Seidel smoothing steps.

This synchronization occurs because of neighbour-to-neighbour communications, required to communicate ghost cells (or halo data) across processor boundaries. Although there is no explicit global communication, and the communication pattern itself is scalable, it is impossible for two processors to become more than one iteration out-of-sync. Consequently, if there is any imbalance between processors, caused by poor load balancing, variable clock frequencies, background tasks, or, most commonly, variability in communication times – then the entire solver is bottlenecked. This is of particular concern as the cells-per-core ratio decreases, and the ratio of communications to computations increases. Likewise, on the coarsest grids of a multigrid solver, this implicit synchronization limits scalability.

In order to remove this implicit synchronization from Jacobi-like solvers or smoothers, one can use the concept of 'Chaotic Relaxations' [10] – a method in which individual rows of the equation-system can be iterated in any chaotic order with guaranteed convergence. This theory has been used to create highly-asynchronous Jacobi-like solvers, such as in Anzt et al. [11]; but this theory can be leveraged further. Here, a completely non-deterministic, chaotic solver is developed in which independent threads are used to overlap computational and communicational work – completely desynchronizing parallel processes and removing a significant scalabililty bottleneck.

In section 2 the theory of chaotic relaxation is introduced and the implementation of this chaotic solver are presented. The solver is tested within the context of a SIMPLE-based CFD solver (ReFRESCO). The chaotic solver is not expected to outperform state of the art solvers, but the benefits and drawbacks compared to the Jacobi method will be evaluated. In section 3 these chaotic methods are applied to a multigrid solver. The objective is to replace Jacobi smoothers with a chaotic smoother, but in order to remove implicit synchronization between levels a novel 'chaotic-cycle' is required to replace normal V-, W- or F-cycling.

The chaotic methods discussed in this paper have been written as a standalone library, *Chaos*[1]. The *Chaos* library is available under an open-source MIT license, in the hope that the results herein can be reproduced, developed and applied to other scientific codes and disciplines. The library is written in C++ and can be bound to Fortran and C easily. Bindings are also created for Python and Ruby automatically, using SWIG[12]. *Chaos* is a hybrid-parallel library, using MPI and OpenMP, and has been tested under multiple compilers and multiple operating systems. The interface is similar to other linear-solver toolkits, such as PETSc [13] and Trilinos [14].

2. Chaotic Relaxation

The task of the linear solver is to solve $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is the unknown solution vector, $\mathbf{A}$ is an N-by-N sparse coefficient matrix, $\mathbf{b}$ is the constant right-hand-side (RHS) vector, and N is the number of elements. Beginning with an initial guess for $\mathbf{x}$, the system can be solved iteratively:

$$\mathbf{x}^k = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{k-1} + \mathbf{D}^{-1}\mathbf{b} \tag{1}$$

where $\mathbf{D}$ is the diagonal of $\mathbf{A}$, and $\mathbf{L}/\mathbf{U}$ are the lower- and upper-triangles respectively. The notation k represents the iteration number. This is the Jacobi method, and the matrix $\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ is the Jacobi iteration matrix, $\mathbf{M}$. Each equation (from 1 to N) can be solved (*a.k.a.* relaxed) independently as follows:

$$x_i^k = \left(-\sum_{\substack{j=1 \\ j\neq i}}^{N} a_{ij} x_j^{k-1} + b_i \right) / a_{ii}, \quad i = 1, \ldots, N. \tag{2}$$

where a, x and b are the individual components of $\mathbf{A}$, $\mathbf{x}$ and $\mathbf{b}$ respectively. At the end of each iteration the new values of $\mathbf{x}$ must be globally communicated before the next iteration can begin. Numerically, improvements to this scheme can be made by using values from the current iteration (k) as they are available. This is the Gauss-Seidel method, and is rarely used due its strictly sequential operation (since the equations must be performed in order) [15]. It is possible to prove the convergence of a parallel block Gauss-Seidel methods, where Gauss-Seidel is performed on local processes and remote updates are iterated in a Jacobi-like fashion – thus an arbitrary selection of variables from iterations k and $k-1$ are used. All of these methods are *stationary* methods, so-called because their formulation does not change between iterations. Additionally, they all have an implicit synchronization point, because no two processes can become more than a single iteration out of sync.

In 1969, Chazan and Miranker [10] proposed the concept of *Chaotic Relaxation* which proves convergence when relaxations are performed out of sync. Each relaxation uses the values of $\mathbf{x}$ from the latest iteration that is available – this could be several iterations behind the current relaxation iteration ($s = 1,\ldots,$n), or even ahead of it ($s < 0$):

$$x_i^k = \left(-\sum_{\substack{j=1 \\ j\neq i}}^{N} a_{ij} x_j^{k-s} + b_i \right) / a_{ii}, \quad s < s_{max}, \quad i = 1, \ldots, N. \tag{3}$$

The order in which the N equations are relaxed is completely arbitrary. With this scheme, processes never need to wait for each other at the end of an iteration. Although communication must still occur, it can be

[1]https://bitbucket.org/jamesnhawkes/chaos

entirely desynchronized, thereby making efficient use of memory bandwidth and computational resources. Processes may even iterate multiple times on the same data if communications are completely saturated, making the best use of the available hardware. Whilst this method is based on the stationary Jacobi method, s can vary between iterations implying that chaotic methods are actually non-stationary.

Chazan and Miranker proved that this iterative scheme will converge for any real Jacobi iteration matrix if $\rho(|\mathbf{M}|) < 1$ so long as s_{max} is bounded. $\rho(\cdot)$ denotes the *spectral radius* (the absolute value of the maximum eigenvalue) and $|\cdot|$ represents a matrix where all the components have been replaced with their absolute values. The implications of s_{max} being bounded is simply that if two relaxations take different amounts of time (either due to imbalanced hardware or relaxation complexity), they cannot be left completely independent indefinitely, such that s could potentially become infinite. Baudet [16] went on to prove that $\rho(|\mathbf{M}|) < 1$ is a necessary condition for convergence for any $s_{max} \leq k$. Bahi [17] further showed that $\rho(|\mathbf{M}|) = 1$ is also valid, if $\mathbf{M}$ is singular and s_{max} is bounded.

For the pressure Poisson equation it can be shown that $\rho(|\mathbf{M}|) \leq 1$ [6], but it is often sufficient to show that the matrix $|\mathbf{A}|$ is diagonally dominant.

At their conception, chaotic methods were considerably ahead of their time. Although created specifically for parallel computing, the concurrency of state-of-the-art supercomputers in 1969 was too small to utilize the methods efficiently. With new architectures, the true potential of chaotic methods may be realized.

The extent to which chaotic relaxation is exploited varies significantly in the literature, and most commonly the exact implementation is not discussed. In Anzt et al. [11] the theory is used to create a 'block-asynchronous' Jacobi solver for GPUs, where a fixed number of local iterations are performed between communications. These solvers are often referred to as *asynchronous* methods, such that *async(5)* refers to a Jacobi method with communications every 5^{th} iteration. This reduces implicit synchronization and shows good results compared to stationary methods, demonstrating that relaxations on 'old' data are not wasted. However, this method could also needlessly throttle communications, thus reducing convergence rates if communication hardware is not saturated. Anzt et al. focuses on GPU architectures, where this may not be such a problem.

In this paper, chaotic relaxation theory is leveraged further. By separating communications and computations into shared-memory threads, the fixed asynchronicity can be removed and the respective hardware can be saturated, leading to increased efficiency and numerical performance. In this truly chaotic method, computation threads never need to wait for communications to be complete and communication hardware is never wasted waiting for computations. Furthermore, a multi-threaded model can exploit access to shared memory, inserting updated values (from communications) directly into the active relaxations, which should give numerical advantages. The targeted architecture for this 'chaotic solver' is a hybrid-parallel (MPI + OpenMP) CPU environment; but it would be possible to extend these chaotic methods to heterogenous environments with relative ease. The main disadvantage of this chaotic solver is that it is non-deterministic, which prevents perfect reproducibility of the solution process – this should not affect the flow solution. The following sections describe the implementation and testing of a chaotic solver.

2.1. Implementation

Consider the matrix $\mathbf{A}$, partitioned contiguously across multiple MPI domains, such that each process owns a block of n rows of length N. It also owns the corresponding portion of the vectors $\mathbf{x}$ and $\mathbf{b}$. To perform a chaotic relaxation on this block, as in equation 3, the off-diagonal elements of the matrix are multiplied by the solution vector. Where these off-diagonal elements correspond to the local portions of $\mathbf{x}$ this is trivial. For non-local elements, the value of $\mathbf{x}_j$ must be obtained from another process via MPI communications.

For the purposes of implementation it is helpful to store the partitioned matrix as two compressed-row-storage (CRS) matrices, $\mathbf{A_A}$ and $\mathbf{A_B}$, as shown in figure 1. $\mathbf{A_A}$ is an n-by-n square matrix (containing the diagonal) which can be directly multiplied by $\mathbf{x_A}$, the local portion of $\mathbf{x}$. $\mathbf{A_B}$ contains all the remaining off-diagonal elements, which must be multiplied by off-process elements $\mathbf{x_B}$. The structure of $\mathbf{A_B}$ and $\mathbf{x_B}$ are re-arranged so that $\mathbf{x_B}$ becomes a local, sparse representation of the parallel vector.

The process of updating this buffer is known as vector-scattering. Each process must pack (only) the required local values of $\mathbf{x_A}$ into an a buffer, which is sent to an neighbouring process via an MPI message, and inserted directly into $\mathbf{x_B}$.

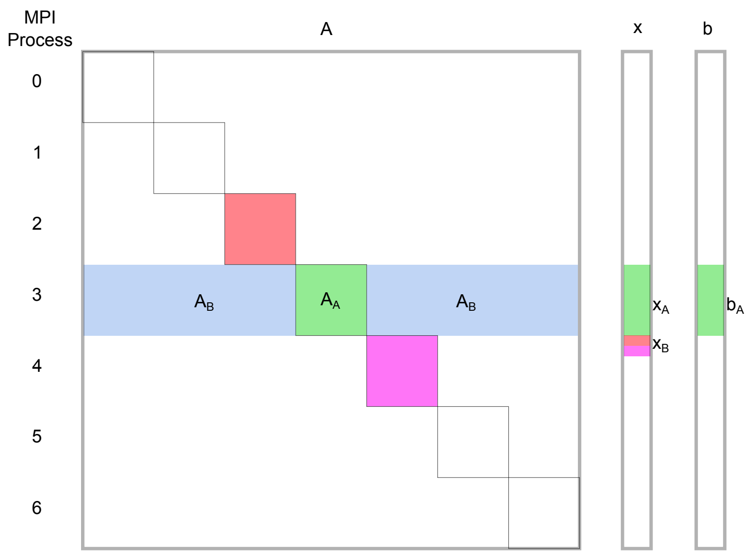


Figure 1: An illustration of the MPI parallel partitioning scheme for the matrix $\mathbf{A}$, and vectors $\mathbf{x}$ and $\mathbf{b}$. Note that the column indices of $\mathbf{A_B}$ are altered to point to the correct location in $\mathbf{x_b}$, the vector-scatter buffer.

In a classical Jacobi solver, it is common to perform part of the relaxation with $\mathbf{A_A}$ and $\mathbf{x_A}$ whilst the scatter is being performed, and complete the iteration using $\mathbf{A_B}$ and $\mathbf{x_B}$ when the communication is complete. In asynchronous methods, vector-scattering can be overlapped over a fixed number of iterations and old values of $\mathbf{x_B}$ are used in the meantime.

Here, we consider a truly chaotic solver in which scattering and relaxing occurs simultaneously on independent threads, and the relaxation uses the most up-to-date values of $\mathbf{x_B}$ that are available. To achieve this, a hybrid-parallel scheme is used. The matrix is still partitioned into MPI subdomains, but within each MPI process several shared-memory OpenMP threads operate. It is commonly recommended to run such a system with each MPI process occupying a single socket (or NUMA-node) of the supercomputer so that all threads are using the same memory channels (avoiding 'Non-Uniform Memory Access' problems). Each OpenMP thread is then given one physical core. For example, a problem previously using 64 MPI processes may be replaced by 8 MPI processes, each running 8 threads.

On each MPI process, one thread is designated as the *communications* thread and the remainder are assigned to *computation*.

The computation threads have shared-memory access to all of $\mathbf{A_A}$, $\mathbf{A_B}$, $\mathbf{x_A}$ and $\mathbf{x_B}$ and continuously perform relaxations on this data. Each thread takes a contiguous portion of the n rows and relaxes them in order, and repeats until the stop criteria are reached. It is not necessary to synchronize the computation threads, so each thread may perform a different number of iterations. Values of $\mathbf{x_A}$ are pushed and pulled from local memory without synchronization or mutexing. Values of $\mathbf{x_B}$ are pulled from local memory too, whilst the communication thread updates them.

The role of the communication thread is to continuously perform vector-scatter operations between MPI processes. It is responsible for collecting local values $\mathbf{x}_A$ and packaging them into MPI messages – once again, no synchronization or mutexing is required and these values are pulled directly from the active memory of the computation threads. On receiving a message, the incoming data is pushed directly into $\mathbf{x}_B$, immediately making it available for relaxations. In this implementation, a vector-scatter must be completed by all processes before the next can begin. As such, every MPI process will perform the same number of communication iterations and will experience implicit synchronization of communication threads. A more chaotic scheme could be implemented, allowing pairs of processes to communicate faster than other pairs if the hardware allowed it. One-sided MPI communications would be the ideal method to pursue this.

This chaotic solver deliberately exploits race conditions to achieve a high level of asynchronicity. Although no two threads write to the same location in memory simultaneously, it is quite possible that threads will try to read and write the same value at the same time. As long as the data type is atomic, this causes no issues. It is not transparent how the compiler or memory controller deals with cache-coherency when encountering these data races, but this also caused no discernible problems; no explicit cache flushing of the $\mathbf{x}$ vector was required.

At the end of a typical solution process, the communication thread on each MPI process has performed O(1k-10k) communication iterations and each computation thread has performed a varying number of computation iterations (often O(100-1k) out of sync). All of this occurs in the time it takes to run just a handful of iterations of a Krylov Subspace solver such as GMRES.

One of the conditions for convergence of chaotic methods is that s (the difference in iteration-number between relaxations), in equation 3, is bounded. There is no explicit checking of the value of s, because a residual check is performed every 100 communication iterations – providing a sufficient condition for convergence. Once convergence criteria are reached, the communication thread sets a flag signalling all threads to cease. There is no guarantee on the number of iterations that have been performed by each thread.

2.2. Numerical Setup for Performance Experiments

The chaotic solver described above has been implemented in *Chaos*, which has been linked to *ReFRESCO*, a state-of-the-art semi-implicit CFD code. Two test cases have been used to conduct performance experiments: a canonical lid-driven cavity flow (LDCF) and a practical, maritime flow simulation around a ship (KVLCC2). The University of Southampton supercomputer, *Iridis4*, has been used to run these experiments from 8 through to 2048 cores. ReFRESCO, Iridis4 and the two test cases are discussed below.

ReFRESCO is a finite-volume, SIMPLE-based, Picard-linearizing, viscous-flow CFD solver, which solves multiphase, unsteady, incompressible flows with a variety of turbulence, cavitation and volume-fraction models. ReFRESCO is a general-purpose CFD code, with state of the art features such as moving, sliding and deforming grids and automatic grid refinement – but has been verified, validated and optimized specifically for maritime-industry problems. ReFRESCO is similar in formulation to other well-known CFD solvers such as OpenFOAM [18], Star-CCM+ [19] or Ansys Fluent [20]. ReFRESCO uses PETSc [13], which provides Krylov Subspace solvers and an interface to ML [9] (via the Trilinos project [14]). Both PETSc and ReFRESCO are MPI-only, which makes interfacing to *Chaos* difficult, but this is inconsequential for the purposes of the following investigation. ReFRESCO is profiled with Score-P [21], providing run-time metrics on various portions of the code. The results herein show only the time spent in the linear equation-system solver for the pressure equation.

ReFRESCO is run on the University of Southampton's latest supercomputer. Iridis4 has 750 nodes, consisting of two Intel Xeon E5-2670 Sandybridge processors (8 cores, 2.6 Ghz), for a total of 12,200 cores (8 cores per NUMA-node). Each 16-core node is diskless, but is connected to a parallel file system, and has 64GB of memory. The nodes run Red Hat Enterprise Linux version 6.3. Nodes are grouped into sets of 30, which communicate via 14 Gbit/s Infiniband. Each of these groups is connected to a leaf switch, and inter-switch communication is then via four 10 Gbit/s Infiniband connections to each of the core switches. Management functions are controlled via an ethernet network.

Iridis4 ranked #179 on the Top500 list of November 2013 with a peak performance of 227 TFLOPS [22]. Iridis4 cannot be classified as a many-core machine, with only 16 cores per node. Nonetheless, it should be able to give sensible insight into the limitations of the CFD algorithm with an appropriately-sized problem. A one-time opportunity to perform experiments up to 2048 cores was provided for this study, but all development was done on up to 512 cores.

Two test cases are used throughout. The first is a laminar-flow, canonical, unit-length, three-dimensional lid-driven cavity flow (LDCF) with a Reynolds number of 1000. A uniform mesh of $N = 2.68$-million cells (139^3) is used. The simulation mimics an infinite domain with two cyclic boundary conditions. The remaining four boundaries are constrained with Dirichlet boundary conditions, one of which specifies a tangential, non-dimensional velocity of 1. The second test case is an industrial model ship case, the KRISO Very Large Crude Carrier (KVLCC2) double-body wind-tunnel (water phase only) model [23] with a Reynolds number of 4.6×10^6. The mesh is a three-dimensional multi-block mesh consisting of $N = 2.67$-million cells. A k-ω, two-equation shear stress transport turbulence model is used. Inflow and outflow boundaries are specified appropriately. The free-surface is represented as a Neumann boundary condition, and the remaining boundaries have Dirichlet conditions. The mesh sizes have been chosen deliberately to show scalability issues on the available cores: the cells-per-core ratio is approximately 1300 on 2048 cores for both test cases.

2.3. *Performance Experiments*

The chaotic solver is compared to a simple MPI-only SOR method from PETSc, with residual calculations every 100 iterations and $\omega = 1$ (equivalent to block Gauss-Seidel); and a Flexible-GMRES solver with right-side Block Jacobi preconditioning, also from PETSc. A scalability factor S can be defined as $S = 8T_8/T_C$ where T_C is the wall-time using C core. Figure 2 shows the results of this scalability study, showing scalability (S) versus the number of cores (C). Ideal scalability is when $S = C$, illustrated in the figure. Due to the normalization of T against T_8, it is not possible to observe absolute performance differences between solvers using scalability plots. The embedded bar charts rectify this, by showing absolute core-hours ($C \times T_C$, which would ideally remain constant) at $C = 8$, $C = 128$ and $C = 2048$, coloured and keyed with respect to the enclosing scalability plot – note the exponential scale. 100 non-linear loops of the SIMPLE algorithm are performed and the total time spent solving the pressure equation is recorded.

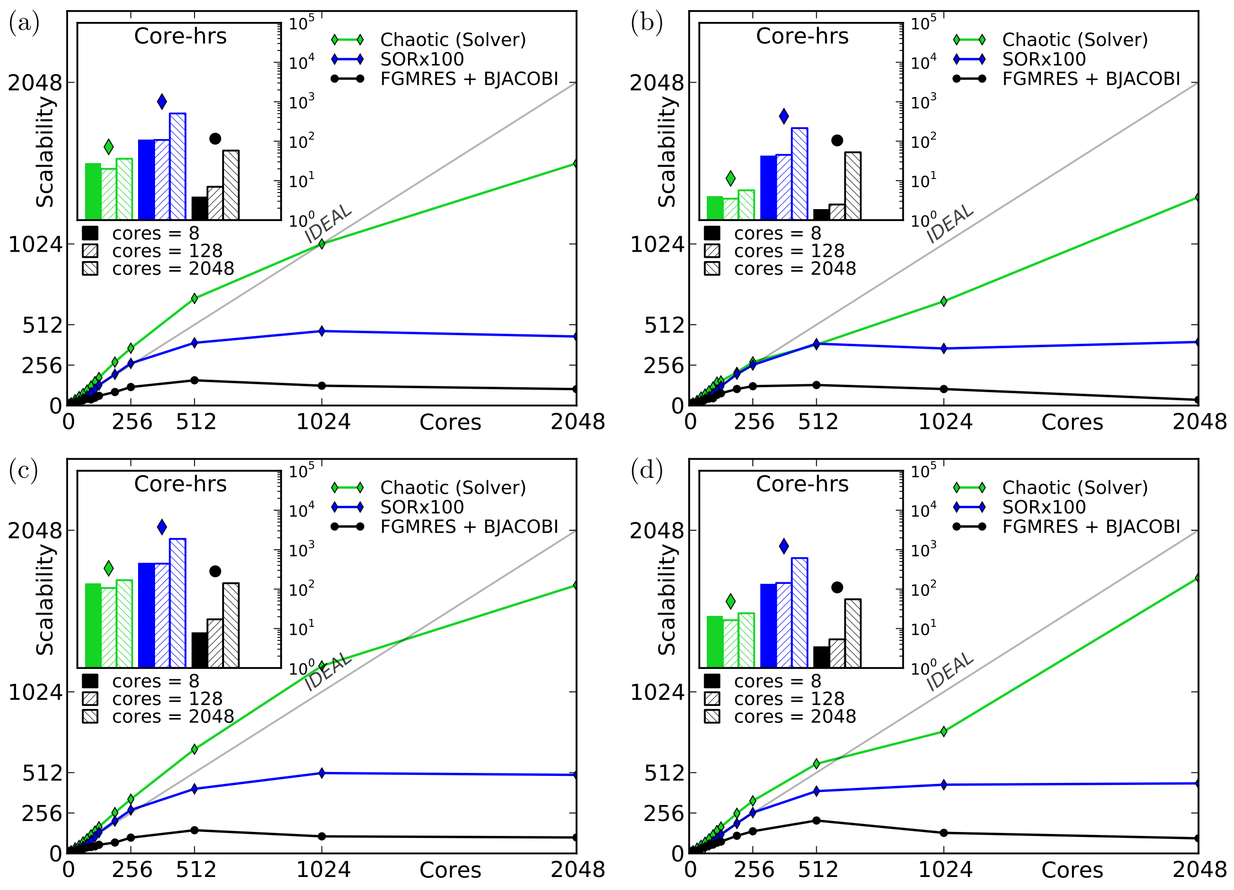


Figure 2: Scalability of the linear equation-system solver for the KVLCC2 (left) and LDCF (right) test cases, at 0.1 (top) and 0.01 (bottom) convergence tolerance.

The LDCF test case appears much easier to solve than the KVLCC2 test case, for all solvers and convergence tolerances. The chaotic solver is faster than SOR in all cases, with the gap widening as the equations become easier to solve. Scalability of the chaotic solver is better than the SOR method, and is sometimes super-linear due to increasing use of the cache as the cells-per-core ratio decreases. Above 512 cores, particularly on the easier equation-systems, the scalability of the chaotic solver begins to deteriorate. Obtaining computing time to thoroughly debug this was not possible, but it is suspected that the residual-check interval needs to be increased at this point. This is partly due to the decreasing cells-per-core ratio; and partly due to the increasing cost of global communications[2]. Although the computation threads continue to iterate whilst waiting for the residual check, the communication thread (and vector-scatter updates) are stalled, reducing convergence rates.

Nonetheless, on all but the hardest equation-system (KVLCC2 at 0.01 convergence tolerance), the chaotic solver is able to out-perform preconditioned FGMRES on 2048 cores, due to the poor scaling of the Krylov Subspace method. On stiffer equation systems, the gains are smaller because more low-frequency errors must be reduced – the numerical performance of SOR and the chaotic solver is not competitive. The two test cases used here are small, designed to show scalability limits on a machine that does not have a many-core architecture. Practical simulations may be tens or hundreds of times larger than these tests, and the inability of SOR or the chaotic solver to reduce low-frequency errors will become much more problematic. The work required by Jacobi-like solvers increases with $O(N^3)$ [24, §2.2][3]. However, the ability to reduce high-frequency errors quickly is very desirable for smoothers in multigrid methods. In the following section, a chaotic-cycle multigrid is developed which allows Jacobi-like smoothers to be replaced with chaotic smoothers.

3. Chaotic-Cycle Multigrid

The principle of algebraic multigrid methods is that low-frequency errors can be solved quickly by translating the problem onto a coarser grid, where low-frequency errors become high-frequency errors. This coarsening can be applied recursively over many levels. Jacobi-like smoothers can be used to quickly smooth high-frequency errors on the coarse grids. Often, only a single iteration of the smoother is required. It can be shown that the number of multigrid iterative cycles required to converge a perfect multigrid method remains constant as N increases, which makes them competitive against Krylov Subspace methods and relevant for real-world applications. In theory, the work done in each multigrid iteration scales with N, since all of the relevant routines (such as smoothing) are $O(N)$. Thus, a multigrid solution should take $O(N)$-time, although this does not include the time taken to construct the coarse grid operators [24, §2.4].

There are certain traits of the chaotic solver which make it difficult to substitute as a smoother in a multigrid method: it is difficult to impose stopping criterion; and guarantee the boundedness of the asynchronicity s. In the chaotic solver, a sufficient condition for convergence was provided by checking the residual – which was necessary anyway. For a multigrid method it is not necessary to compute a residual on the coarse grids, and it would be inefficient to do so, so an alternative method for bounding s is required.

Another issue is that smoothers only need to run for a small number of iterations, which does not suit chaotic methods well, especially when the equation-systems are small and easy to solve (as on the coarse grids). Chaotic solvers have a tendency to over-converge these systems considerably, because the computation threads achieve so much smoothing before a single communication can even occur. Although they can never be slower than existing smoothers because of this, much of this free convergence is wasted if it not required. Since there are other sources of synchronization in the multigrid cycle, it would be beneficial to trade off some of these synchronizations for additional smoothing, which can be obtained for free.

[2]The cost of a global reduction, required for a residual-norm computation, usually scales with $\log_2(C)$ – increasing each time the number of cores is doubled. However, this assumes the latency of communications is constant. When performing global reductions on 2048 cores, one encounters higher latencies as the cores become physically further apart.

[3]Note that the rate of convergence increases with $O(N^2)$, but the work-done-per-iteration also increases with $O(N)$, leading to overall required work of $O(N^3)$.

A generic multigrid algorithm has been developed below, employing an unsmoothed-aggregation technique and the classical V-, W- and F-cycles. Following this, the extension to a 'chaotic-cycle' multigrid is explained, where the boundedness of s and the removal of other synchronization points is explored. Verification of $O(N)$ performance is shown on a generic Laplace equation. Scalability and performance of all the multgrid cycles are tested against the standard chaotic solver, FGMRES and a state-of-the-art multigrid method (ML) from Trilinos [9, 14].

3.1. Implementation

There are two stages to a multigrid algorithm. The first stage is concerned with constructing coarse grids and interpolation operators. The second stage is the multigrid 'cycle' which is the iterative process used to solve the equation system. In the coarsening stage, coarse grids are recursively created from a fine grid, such that:

$$\mathbf{A}_m = \mathbf{R}_m \mathbf{A}_{m-1} \mathbf{P}_m, \quad 1 \leq m \leq m_{max}$$

where $\mathbf{A}_m$ is matrix $\mathbf{A}$ on the m^{th} level (level zero being the finest, original matrix); $\mathbf{P}_m$ is the prolongation matrix, a sparse rectangular matrix which describes the interpolation between $\mathbf{A}_m$ and $\mathbf{A}_{m-1}$; and $\mathbf{R}_m$ is the restriction matrix, a sparse rectangular matrix describing the contraction from $\mathbf{A}_{m-1}$ to $\mathbf{A}_m$. $\mathbf{R}_m$ is usually the transpose of $\mathbf{P}_m$.

Here, an aggregation-type coarsening is used, loosely based on Notay [25]. In aggregation schemes, a coarse grid is constructed by grouping fine-grid elements into single coarse-grid elements (or 'aggregates'), such that the aggregates form a disjoint subset of the fine-grid elements. In unsmoothed aggregation, $\mathbf{R}_m$ and $\mathbf{P}_m$ are simply boolean matrices which never need to be explicitly stored. In smoothed aggregation, the prolongation and restriction matrices are smoothed such that coarse-grid elements may contribute to more than one fine-grid element. The simpler unsmoothed-aggregation is used here.

When constructing the coarse grids, it is important that aggregates are created in the direction of the 'smoothest error'; they must be constructed such that the maximum eigenvalue is successfully reduced on the coarser grid [26]. Notay [25] proposes a pairwise-aggregation scheme which attempts this. The algorithm couples pairs of elements which are 'strongly-coupled', which means they have large negative values in the off-diagonals that connect them. For each fine-grid element a list of q strongly-coupled elements is created. The element with the smallest q is treated first, by grouping it with its most-strongly-coupled pair. For asymmetric matrices a check can be added to ensure that the coupling is reciprocated, but this may be skipped. Constantly sorting to find the minimum q can be expensive, and a pairwise scheme can only coarsen by a factor of two – meaning that repeated application is necessary to achieve 4- or 8-times coarsening.

A cheaper, but less rigorous approach has been adapted from this scheme. A maximum aggregation size a_{max} is specified (for example, 8). For each element e_i in the fine grid (iterated in order), the off-diagonal values of the matrix $(\mathbf{A}_{m-1})_{ij,j\neq i}$ provides a set of coupled elements e_j that can form aggregates. For each coupled element, the strength of the coupling is recorded as $v_j = -(\mathbf{A}_{m-1})_{ij}$. If the coupled element already belongs to an aggregate, the number of elements already in that aggregate is recorded as a_j, else $a_j = 1$. An aggregate is formed by selecting e_j with a minimal a_j, and then a maximal v_j. The aggregation of the two elements is abandoned if both elements already belong to aggregates, or if either element belongs to an aggregate which already has a_{max} elements. The algorithm achieves close to a_{max} coarsening in a single step, very cheaply.

At the end of the aggregation process, a set of aggregates $G_i, i = 1, ..., n_m$ exist, each with approximately a_{max} indices which map to $\mathbf{A}_{m-1}$. This mapping is equivalent to $\mathbf{R}_m$ and the tranpose mapping is created for $\mathbf{P}_m$. From this point, $\mathbf{A}_m$ can be computed recursively as:

$$(\mathbf{A}_m)_{ij} = \sum_{k \in G_i} \sum_{l \in G_j} (\mathbf{A}_{m-1})_{kl} \quad (1 \leq i, j, \leq n_m)$$

The process is repeated recursively, with $m = m + 1$, until the coarsest grid reaches a specified size. Figure 3 shows an example of this coarsening on a two-dimensional lid-driven cavity flow domain. Investigating

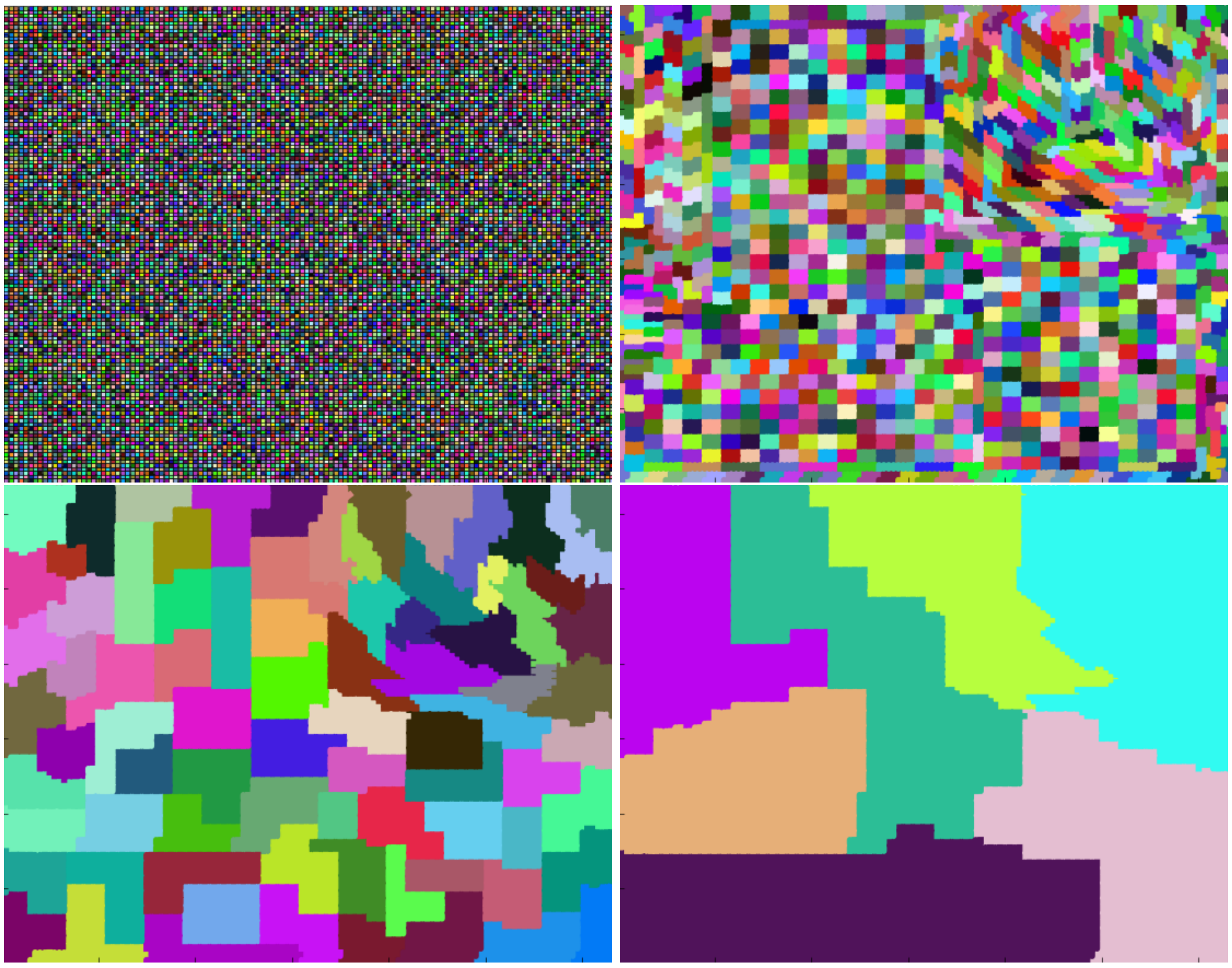


Figure 3: A view of the aggregation process, showing four multigrid levels with $a_{max} = 8$ on a two-dimensional lid-driven cavity flow domain. The fine matrix begins with $128 \times 128 = 16384$ degrees of freedom; and the coarsest matrix has just 7.

the quality of the aggregation is beyond the scope of this study, but the presented method appears to be robust enough for the purposes of these experiments.

The aggregation is done per MPI domain, such that each process has its own set of hierarchical grids (as in [25]). Aggregation is not performed across process boundaries, thus $\mathbf{A}_m$ in the above discussion is replaced by $(\mathbf{A}_A)_m$ for each MPI domain. $\mathbf{A}_B$ is never considered when aggregating. Also, when constructing the coarse grids, $(\mathbf{A}_B)_m$ is not explicitly constructed because it would require re-mapping the buffered storage of the parallel vectors, $(\mathbf{x_B})_m$. Vector-scattering on coarse grids occurs by mapping into the buffer of the finest grid, resulting in excess, inefficient MPI communication. A more rigorous aggregation scheme which performs coarsening across MPI domains may be more performant; especially since this aggregation can be reused between non-linear iterations of the overall CFD simulation.

The second stage of the multigrid algorithm is the linear solution process, in which various different cycles can be used to visit the coarse grids. On each coarse grid, a 'correction' to the fine-grid solution is computed using the additive correction scheme [27]. There are three simple routines which allow a variety of cycles to be created.

- On each level, smoothing is applied to the linear system $\mathbf{A}_m\mathbf{x}_m = \mathbf{b}_m$. On the coarsest level a direct solver is often used, but a smoother will suffice too.

- Restriction is used to move from a fine grid $(m-1)$ to a coarse grid (m): from the linear system $\mathbf{A}_{m-1}\mathbf{x}_{m-1} = \mathbf{b}_{m-1}$, a new system is created $\mathbf{A}_m\mathbf{x}_m = \mathbf{b}_m$. $\mathbf{A}_m$ has been generated by the aggregation process, $\mathbf{x}_m$ is set to zero, and $\mathbf{b}_m$ is set to to the residual vector multiplied by the restriction matrix: $\mathbf{R}_m(\mathbf{b}_{m-1} - \mathbf{A}_{m-1}\mathbf{x}_{m-1})$.
- In the other direction, prolongation is used to apply the coarse-grid correction to the fine grid. The prolongation matrix, $\mathbf{P}_{m-1}$ is used to expand $\mathbf{x}_m$ into $\mathbf{x}_{m-1}$. The vector $\mathbf{P}_m\mathbf{x}_m$ is summed with $\mathbf{x}_{m-1}$, to provide a *correction* to the fine grid. Post-smoothing is performed after prolongation.

These three routines, combined with different values of a_{max}, different minimum grid sizes, and different numbers of smoothing steps, can produce a wide variety of multigrid cycles. A four-level V-cycle, F-cycle, W-cycle and sawtooth-cycle are illustrated in figure 4.

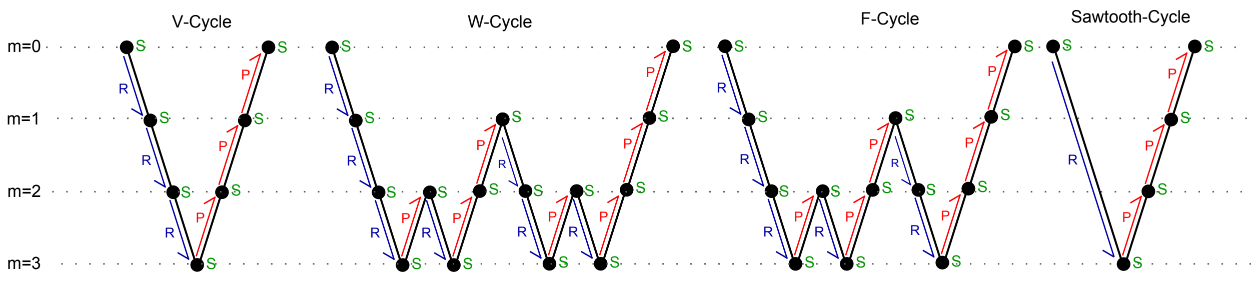


Figure 4: A view of the solution process, showing a single iteration of a four-level multigrid method using the classical recursive V-cycle, W-cycle, F-Cycle and Sawtooth-Cycle. Restriction, Prolongation and Smoothing are colour-coded.

3.2. Applying Chaotic Principles to Multigrid Methods

Consider the classical V-cycle as illustrated. In each pre-smoothing step there is an implicit global synchronization during each iteration, since MPI communications are required to scatter the $\mathbf{x}_m$ vector on each level. For restriction, a vector-scatter is necessary to compute $\mathbf{b}_m = \mathbf{R}_m(\mathbf{b}_{m-1} - \mathbf{A}_{m-1}\mathbf{x}_{m-1})$, causing another global implicit synchronization.

Replacement of the smoother with a chaotic smoother solves the implicit synchronization in the pre-smoothing steps, but since the smoother only needs to run for a handful of iterations, the synchronization required for restriction will still have a large effect. In order to minimize this, consider a sawtooth cycle, which is simply a V-cycle with no pre-smoothing – the right-hand sides $(\mathbf{b}_m)$ of the all the coarse matrices must still be initialized. In the 4-level example shown in figure 4, communication of $\mathbf{x}_0$ is required as usual to create $\mathbf{b}_1$, requiring a vector-scatter. $\mathbf{x}_1$ is reset to zero. The next restriction follows immediately to create $\mathbf{b}_2$; except now it is known that $\mathbf{x}_1 = 0$ there is no need to scatter $\mathbf{x}_1$, and $\mathbf{b}_2 = \mathbf{R}_2(\mathbf{b}_1 - \mathbf{A}_1\mathbf{x}_1) = \mathbf{R}_2\mathbf{b}_1$. It follows that the entire sawtooth restriction requires only one implicit synchronization. The downside of using a sawtooth method is that more post-smoothing iterations may be required. For the chaotic smoothers this may not be a disadvantage, because much of this smoothing can be obtained for free.

Returning from the coarse grid to the fine grid, post-smoothing with a chaotic smoother does not require any implicit synchronization. Prolongation does not require any synchronization either; because $\mathbf{x}_{m-1} \mathrel{+}= \mathbf{P}_m\mathbf{x}_m$ does not require off-process values of $\mathbf{x}_m$. Smoothing on level $m-1$ can begin before other processes have performed their own prolongation, because the off-process values of $\mathbf{x}_m$ that are not updated are still zero. Updated values will be provided by communications in the chaotic smoother when they are ready, but chaotic relaxations can begin without that information.

Next, the boundedness of s in the chaotic smoother must be guaranteed. Consider the situation in which p post-smoothing iterations are requested, and this is expected to provide enough convergence on the coarse grids to guarantee overall convergence of the multigrid method. Note that a certain amount of post-smoothing must be performed in order to compensate for unsmooth prolongation [24]. For the chaotic smoother, specifying a number of iterations is meaningless because one process could complete its iterations before receiving updates from any other process or thread.

The chaotic-cycle, like the chaotic solver, is designed to be run in a hybrid MPI+OpenMP environment. The same concept of computation and communication threads is used, and the communication threads remain in sync across MPI processes. Within each MPI process, each thread stores an iteration counter k in an array, initialized to $k = 0$ at the start of each smoothing phase. After each computation thread completes a chaotic relaxation on its portion of the equation system, it attempts to increment k – however, it is only allowed to increment k if the values in the counter array are all $\geq k$. If any value is less than k, it suggests that the relaxations may have used old values and therefore it did not 'count' as a requested iteration. This bounds s between computation threads. The communication thread also participates in the same counter array, attempting to increment k every time it scatters $\mathbf{x}_m$ between MPI processes; thus bounding s between all processes. To avoid an MPI deadlock, additional checks are required to make sure communication threads perform the same number of iterations on all processes (regardless of k); this is done using non-blocking MPI barriers (MPI_Ibarrier).

It is unlikely that any thread will complete exactly p iterations; but it is the absolute lower bound. Most threads will complete many more than p iterations, so choosing a value for p is not as intuitive when compared to classical multigrid cycles.

Combining this bounded chaotic smoother with the sawtooth cycle and barrierless prolongation: a chaotic-cycle multigrid is created. The entire cycle has only one implicit global synchronization, required to restrict the finest level. Instead of explicitly scattering $\mathbf{x}_0$ the latest values communicated during the pre-smoothing step are used, further hiding this implicit synchronization behind useful relaxations. One explicit thread-barrier (local) is also required at the end of the restriction simply to signal that all of the coarse grids have been initialized, but this is cheap. It is important that $\mathbf{x}_m$ has been reset to zero on every level before prolongation begins. During each multigrid cycle, a lagged, non-blocking residual is computed to determine an overall stoping criteria.

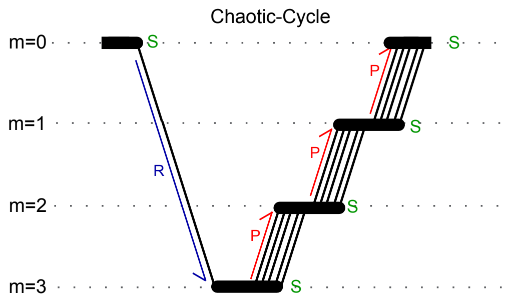


Figure 5: A view of the chaotic-cycle solution process, showing a single iteration of a four-level multigrid method. Restriction, Prolongation and Smoothing are colour-coded. Multiple parallel lines are shown during the prolongation stage to depict the non-deterministic way in which multiple threads (including communication and computation threads, across multiple processes) may participate in the chaotic-cycle.

Figure 5 shows an illustration of the chaotic-cycle, although it is difficult to capture the chaotic nature in a simple diagram.

3.3. Verification

In order to ascertain that the chaotic-cycle conforms to the convergence expectations of a multigrid method, verification is performed using a 3-dimensional (i, j, k) Laplace equation with Dirichlet boundaries. A 3D, unit-length, cubic domain of N elements is created (element spacing $h = 1/N^{\frac{1}{3}}$). A pre-determined solution $\mathbf{x}_s$ is created for the linear system, by combining 20 sinusoidal frequencies: $\sum_{f=0}^{20} \sin(2^f \pi.i.j.k.h^3)$. $\mathbf{b}$ is created from this solution and $\mathbf{x}$ is set to zero. The source is included as an example in *Chaos*.

The number of iterations to convergence (relative tolerance 10^{-6}) and time-per-iteration is measured for the chaotic-cycle and V-cycle as N increases from 512 (8^3) to 1.73-million (120^3). With perfect coarsening, the number of iterations to convergence should be constant, with the time-per-iteration increasing linearly with N. However, this ideal situation cannot be achieved with piecewise-constant unsmoothed aggregation [26]. Figure 6 shows the results of this verification. The solution is performed on 64 cores (8 MPI processes with 8 threads each). The aggregation factor, a_{max}, is set to 8, and the number of pre- and post-smoothing iterations is set to $p = 3$.

The number of iterations required to reach convergence is not quite constant, but appears to approach an asymptotic maximumum, for both cycles. This suggests small imperfections in the aggregation process, which is expected. The time-per-iteration is $O(N)$ as desired, although both the V- and chaotic-cycle show occasional deviations or noise. This is probably connected to the aggregation factor, and the number of levels being created as N increases.

Overall, the two cycles perform similarly; but these tests are not designed to give performance results. The important result here is that the chaotic-cycle is at least as good as the V-cycle from the perspective of multigrid correctness. The following section properly assesses the performance and scalability of the various cycling techniques.

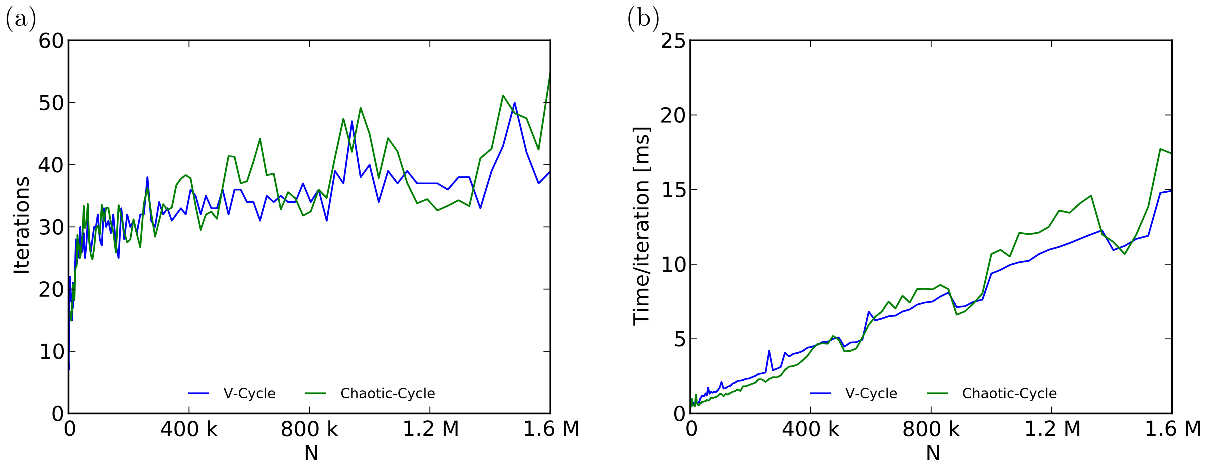


Figure 6: Number of iterations (a) and time-per-iteration (b) to converge the discrete Laplace equation, of size N, to a relative convergence tolerance of 10^{-6}.

3.4. Performance Experiments

The performance experiments from section 2.3 are repeated here, this time comparing the chaotic-, V-, W- and F-cycle with preconditioned FGMRES, the standard chaotic solver, and a state-of-the-art multigrid package (ML [9, 14]). Once again the solution of the pressure equation is summed over 100 outer-loops of the CFD solver. For all multigrid methods, including ML, aggregation is only performed once in this time, and then re-used. This is possible because the non-zero structure of the matrix $\mathbf{A}$ does not change between outer loops, thus the restriction and prolongation matrices do not need recomputing. $\mathbf{A}_m$ must still be refilled on the coarser grids using equation 3.1. Aggregation and re-filling time is included in the results. Occasional re-aggregation is probably required for practical simulations, because the coupling of various elements may change as the overall CFD solution converges, and the coarser grids will no longer follow the smoothest errors.

Preliminary studies showed that p (the number of pre- and post-smoothing iterations) had little effect on the results within a range of 1 to 12. Increasing p resulted in fewer cycles being required overall (and vice versa), but with almost no effect on scalability and absolute speed. $p = 3$ was selected for all of the *Chaos* multigrid methods; and $p = 1$ was used for ML. ML uses a complex smoothed-aggregation coarsening strategy which crosses process boundaries, thus less smoothing is required. An additive-correction V-cycle was used in ML, with a direct solver on the coarsest grid.

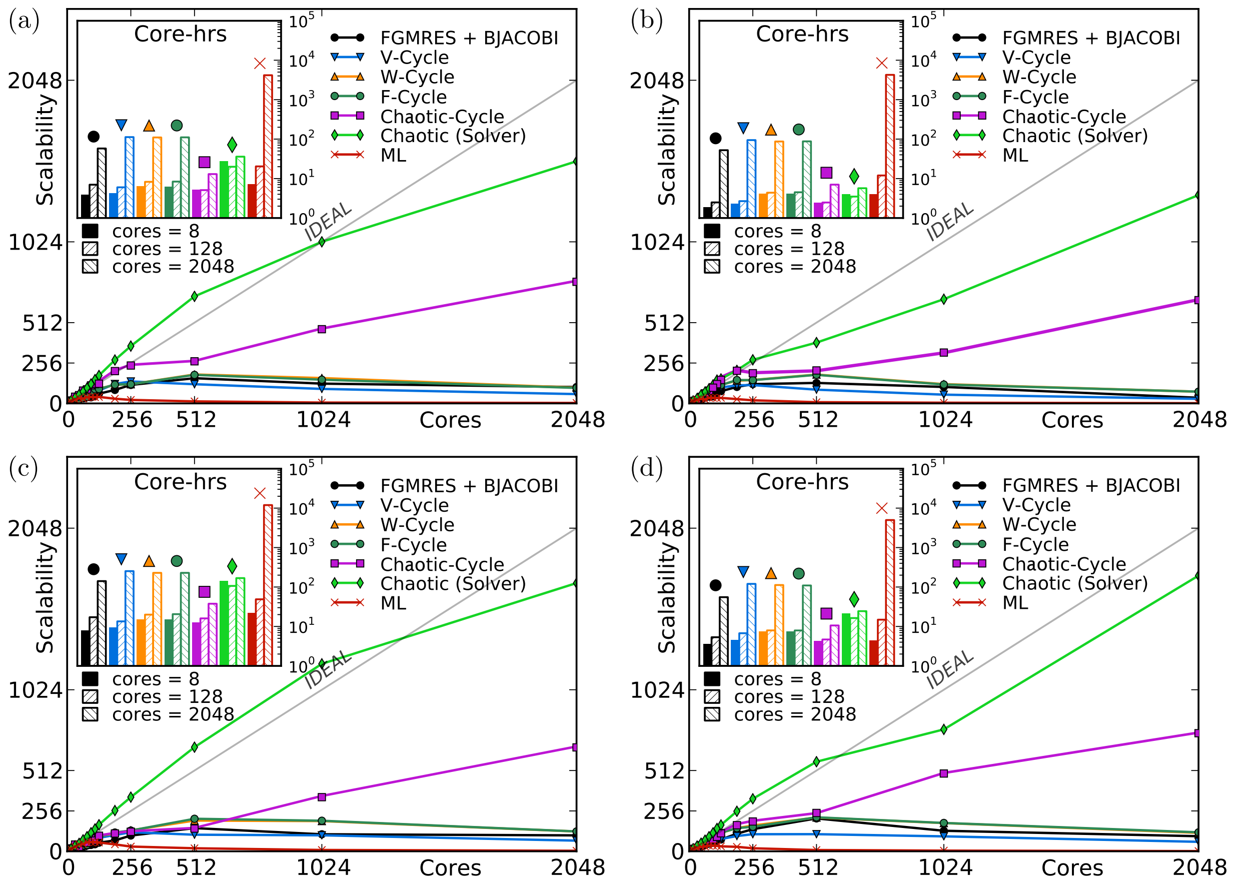


Figure 7: Scalability of the linear equation-system solver for the KVLCC2 (left) and LDCF (right) test cases, at 0.1 (top) and 0.01 (bottom) convergence tolerance.

The results are shown in figure 7 and table 1. Overall, the scalability of the V-, W- and F-cycle multigrid is poor, and somewhat similar to the Krylov Subspace method (FGMRES). The V-, F- and W-cycle all perform similarly on $C = 2048$ cores. On a smaller number of cores, the size of $\mathbf{A}_A$ in each domain becomes larger, thus the total number of levels increases and the algorithmic differences between the three cycles become larger. As the number of levels increases, the W- and F- cycles spend exponentially more time visiting coarse grids, and perform worse than the V-cycle. If only two grids existed, all three cycles would be identical. This effect makes it difficult to differentiate between numerical effects and scalability of the computational methods.

The chaotic-cycle exhibits slightly worse absolute performance on $C = 8$ than the V-cycle and FGMRES, but offers much greater scalability. In all cases the chaotic-cycle is the fastest solver on $C = 128$ cores. On the most difficult test case (KVLCC2 at 0.01 convergence tolerance), the chaotic-cycle is over 6.7× faster

Table 1: Absolute core-hours spent solving the linearized pressure equation over 100 non-linear iterations.

Case (ILCT)	FGMRES			V-Cycle			Chaotic-Cycle			Chaotic (solver)		
	$C = 8$	128	2048	8	128	2048	8	128	2048	8	128	2048
LDCF (0.1)	1.8	2.5	54.4	2.3	2.7	94.6	2.3	2.5	7.1	3.6	3.4	5.6
LDCF (0.01)	3.4	5.3	56.8	4.3	6.8	124.1	4.0	4.7	10.6	20.7	16.9	24.5
KVLCC2 (0.1)	3.8	7.1	60.1	4.1	6.1	117.2	5.1	5.3	13.6	27.2	20.8	36.6
KVLCC2 (0.01)	7.8	17.4	144.0	9.2	13.9	258.9	12.4	16.3	38.5	139.9	111.6	169.6

than the classical V-cycle on $C = 2048$ cores, and 3.7× faster than preconditioned FGMRES. This gap widens on easier equations, reaching a 13.3× and 7.7× speed-up over V-cycle and FGMRES, respectively, on LDCF at 0.1 convergence tolerance.

The scalability of ML was poor in all cases. It was suspected that the rigorous aggregation process is a bottleneck to scalability when such a lax convergence tolerance is specified; however, re-use of the aggregation had little effect on scalability. Other efforts to experiment with ML were unsuccessful in improving scalability – including adjusting the number of multigrid levels, the amount of smoothing, and replacement of the direct solver with a smoother on the coarsest grid.

Despite its good performance, the chaotic-cycle multigrid falls short of ideal scalability. Indeed, in one case the simple chaotic solver is faster than chaotic-cycle multigrid due to its superior scaling.

3.5. Discussion

Both the classical- (V,W,F) and chaotic-cycle should achieve better scalability if the vector-scattering on coarse-grids is improved. Currently, the MPI buffers and indexing of $(\mathbf{x}_B)_0$ and $(\mathbf{A}_B)_0$ are recycled on the coarse grids, which is inefficient. Recomputing buffers for each coarse-grid would certainly improve scalability. It is likely that the chaotic-cycle hides some of these losses (since extra relaxation can occur whilst waiting for these communications), so the differences between the two solvers may become smaller.

Perhaps a bigger concern is (once again) the residual computations which currently occur asynchronously during each multigrid iteration. On 2048 cores the aggregation process only creates four levels, and with $p = 3$ a chaotic-cycle will perform a minumum of 18 vector-scatters in this time. As noticed in the chaotic solver, with a residual computed every 100 iterations this was already the main limitation to overall scalability. It is suspected that this is the main cause of poor scalability and is also the only sensible reason that the standard chaotic solver could out-perform the chaotic-cycle multigrid (since the chaotic-solver generalizes to a one-level chaotic-cycle multigrid method). For both the chaotic-cycle multigrid and chaotic solver, the ideal solution would be to apply a threaded residual computation check, such that residuals are only computed at the speed at which they can be communicated. In this way the residual computation could never stall the communication thread, but it would also never lag unnecessarily. In some ways this could be considered a 'chaotic' residual.

The problem with such a scheme is that it is difficult to stop MPI processes safely without some form of explicit, predictable barrier or residual – because if any process issues MPI_Isend/MPI_Irecv commands before receiving the 'stop' notification, these unmatched communications will cause a deadlock. As mentioned earlier, some form of one-sided communication would be ideal to allow communication threads to operate chaotically. This would also allow chaotic residual computations to be implemented easily.

Profiling the chaotic-cycle multigrid is somewhat difficult because even though it can be seen that coarse-grid communications or residual checks are slow, it is not clear whether this is the main source of performance losses. Indeed, the whole purpose of chaotic methods is to make use of all available hardware, even when some systems (such as communications) are struggling. However, it is suspected that the residual issue is far greater, because whilst waiting for the residual to complete the communication threads are not updating the communication buffers on the finest grid. At least on the coarse grids, $\mathbf{x}_B$ is periodically updated, providing fresh values for the excess relaxations to work on.

It is likely that both of the aforementioned issues affect the V-, W- and F-cycle too. Indeed, since they do not have the benefit of performing chaotic relaxations they may be more affected. Nonetheless, the chaotic-cycle multigrid demonstrates very good scaling compared to these classical cycles; and its ability to weather ineffective communications should allow it to perform well in many-core environments.

4. Conclusion

The work presented has focused on applying the theory of 'chaotic relaxations' to simple Jacobi-like solvers, to create a 'chaotic solver'; and adaptation of classical multigrid cycles to allow effective use of chaotic solvers as smoothers in a 'chaotic-cycle' multigrid method.

The chaotic solver provided excellent performance and scalability from 8 through to 2048 cores compared to a standard SOR solver. The chaotic solver outperformed state-of-the-art solvers in the extremes of strong scalability, but could not provide enough numerical power to compete on stiffer equation systems. Since the work done by SOR and the chaotic solver scales with $O(N^3)$, these solvers are not a sustainable solution and will always be beaten in practical simulations.

The chaotic-cycle multigrid outperforms all other solvers (including Krylov Subspace solvers, classical multigrid methods, and state-of-the-art smoothed-aggregation multigrid methods) in terms of scalability and absolute speed, from 128 cores upwards. Below this, the chaotic-cycle multigrid is still highly competitive. On 2048 cores the chaotic-cycle multigrid is up to 7.7× faster than Flexible-GMRES and 13.3× faster than classical V-cycle multigrid. Scalability of the chaotic-cycle multigrid can still be improved. Indeed, on the easiest equation system the simple chaotic solver was faster (on 2048 cores) and this is indicative of an issue with too-frequent residual checks. $O(N)$ performance of the chaotic-cycle multigrid solver has also been verified.

The theory of chaotic relaxations can be applied in a huge variety of ways to create chaotic solvers and multigrid cycles. There are a number of issues with the current implementation which have been discussed, but the potential of chaotic methods compared to synchronous methods has been clearly demonstrated. The chaotic methods discussed here have been implemented as an open-source library, *Chaos*, in the hope that alternative chaotic schemes can be explored. It is also expected that the chaotic solver and chaotic-cycle multigrid could be applied to other disciplines and scientific fields where fast, scalable solutions to stiff linear systems are required.

Acknowledgements

Thousands of simulations have been performed to obtain the results presented herein, and thousands more during development and preliminary investigations. The authors acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work. In particular, the authors wish to thank Ivan Wolton for his cooperation in running the large 1024/2048-core jobs on Iridis4 – without which many meaningful results would have been unobtainable.

[1] J. Shalf, The Evolution of Programming Models in Response to Energy Efficiency Constraints, *Oklahoma Supercomputing Symposium*, Norman, Oklahoma, USA., 2013.

[2] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, S. Williams, K. Yelick, ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, *DARPA IPTO*, 2008.

[3] S. Horst, Why We Need Exascale And Why We Won't Get There By 2020, *Optical Interconnects Conference*, Santa Fe, New Mexico, USA, 2013.

[4] Top 500 List, `http://www.top500.org`, Acc. 2017.

[5] J. Hawkes, S. R. Turnock, S. J. Cox, A. B. Phillips, G. Vaz, On the Strong Scalability of Maritime CFD, Journal of Maritime Science and Technology .

[6] J. Hawkes, S. R. Turnock, S. J. Cox, A. B. Phillips, G. Vaz, Potential of Chaotic Iterative Solvers for CFD, *The 17th Numerical Towing Tank Symposium (NuTTS 2014)*, Marstrand, Sweden, 2014.

[7] P. Ghysels, T. J. Ashby, K. Meerbergen, W. Vanroose, Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines, Journal of Scientific Computing 35 (1) (2013) 48–71.

[8] X.-Y. Zuo, L.-T. Zhang, T.-X. Gu, An Improved Generalized Conjugate Residual Squared Algorithm Suitable for Distributed Parallel Computing, Journal of Computational and Applied Mathematics 271 (2014) 285–294.

[9] M. Gee, C. Siefert, J. Hu, R. Tuminaro, M. Sala, ML 5.0 Smoothed Aggregation User's Guide, Tech. Rep. SAND2006-2649, Sandia National Laboratories, 2006.

[10] D. Chazan, W. Miranker, Chaotic Relaxation, Linear Algebra and its Applications 2 (2) (1969) 199–222.

[11] H. Anzt, S. Tomov, J. Dongarra, V. Heuveline, A Block-Asynchronous Relaxation Method for Graphics Processing Units, Journal of Parallel and Distributed Computing 73 (12) (2013) 1613–1626.

[12] SWIG, `http://www.swig.org`, Acc. 2017.

[13] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, H. Zhang, PETSc Users Manual, Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory, `http://www.mcs.anl.gov/petsc`, 2013.

[14] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, An Overview of Trilinos, Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.

[15] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, Philadelphia, PA, 1994.

[16] G. M. Baudet, Asynchronous Iterative Methods for Multiprocessors, J. ACM 25 (2) (1978) 226–244.
[17] J. Bahi, Asynchronous Iterative Algorithms for Nonexpansive Linear Systems, Journal of Parallel and Distributed Computing 60 (2000) 92–112.
[18] OpenFOAM, OpenFOAM User Guide, 2014.
[19] CD-adapco, STAR-CCM+, http://www.cd-adapco.com, 2014.
[20] ANSYS, FLUENT, http://www.ansys.com, 2011.
[21] VI-HPS, Score-P, v.1.2.3, http://www.vi-hps.org/projects/score-p, Acc. 2013.
[22] Top 500 List, http://www.top500.org, Acc. 2013.
[23] S. Lee, H. Kim, W. Kim, S. Van, Wind Tunnel Tests on Flow Characteristics of the KRISO 3,600 TEU Container Ship and 300K VLCC Double-Deck Ship Models, Journal of Ship Research 47 (1) (2003) 24–38.
[24] P. Wesseling, An Introduction to Multigrid Methods, Pure and Applied Mathematics, John Wiley & Sons Australia, Limited, ISBN 9780471930839, 1992.
[25] Y. Notay, An Aggregation-Based Algebraic Multigrid Method, Electronic Transactions on Numerical Analysis 37 (6) (2008) 123–146.
[26] A. Napov, Y. Notay, Algebraic Analysis of Aggregation-Based Multigrid, Numerical Linear Algebra with Applications 18 (3) (2011) 539–564, ISSN 1099-1506, URL http://dx.doi.org/10.1002/nla.741.
[27] B. R. Hutchinson, G. D. Raithby, A Multigrid Method Based on the Additive Correction Strategy, Numerical Heat Transfer 9 (1986) 511–537.