Chaotic Multigrid Methods for the Solution of Elliptic Equations

J. Hawkes[a,b,*], G. Vaz[b], A.B. Phillips[c], C.M. Klaij[b], S.J. Cox[a], S.R. Turnock[a]

[a] *University of Southampton, Boldrewood Campus, Southampton, United Kingdom*
[b] *Maritime Research Institute Netherlands (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 capabilities of scientific modelling at a similar rate. However, to utilize these machines effectively for applications such as computational fluid dynamics, improvements to strong scalability are required. Here, the particular focus is on semi-implicit, viscous-flow CFD, where the largest bottleneck to strong scalability is the parallel solution of the linear pressure-correction equation – an elliptic Poisson equation. State-of-the-art linear solvers, such as Krylov subspace or multigrid methods, provide excellent numerical performance for elliptic equations, but do not scale efficiently due to frequent synchronization between processes. Complete desynchronization is possible for basic, 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 converge elliptic equations – even with the relatively lax convergence requirements of the example CFD application. However, these chaotic principles can also be applied to multigrid solvers. In this paper, a 'chaotic-cycle' algebraic multigrid method is described and implemented as an open-source library. It is tested on a model Poisson equation, and also within the context of CFD. Two CFD test cases are used: the canonical lid-driven cavity flow and the flow simulation of a ship (KVLCC2). The chaotic-cycle multigrid shows good scalability and numerical performance compared to classical V-, W- and F-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. Further improvements to chaotic-cycle multigrid can be made, relating to coarse-grid communications and desynchronized residual computations. It is expected that the chaotic-cycle multigrid could be applied to other scientific fields, wherever a scalable elliptic-equation solver is required.

Keywords: Chaotic Cycle, Multigrid, Chaotic Relaxation, Exascale, Strong Scalability, Elliptic Equations

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 began 'leaking' across the dielectric gates, and voltages had to be increased to maintain stability. To keep up with exponential growth expectations, manufacturers have resorted to packaging multiple cores onto a single chip, 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

*Corresponding author
Email address: `j.hawkes@soton.ac.uk` (J. Hawkes)
URL: `https://bitbucket.org/jamesnhawkes/chaos` (J. Hawkes)

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 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 scientific applications 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.

This paper concerns itself with incompressible-flow CFD solvers, which find a solution to the Navier-Stokes equations using the iterative SIMPLE algorithm (Semi-Implicit Method for Pressure Linked Equations) or variants thereof [5, §6.7]. In particular, a viscous-flow, finite-volume code (ReFRESCO[1]) is used. For each SIMPLE iteration, linearized versions of the governing equations for momentum, continuity and other modelled quantities are created and solved. The SIMPLE loop is responsible for updating non-linear terms and coupling these equations. In this case, linearization is performed using the Picard method; the continuity-equation is rearranged as a pressure-correction equation (or just 'pressure equation'); and the equations are solved in a segregated manner [6]. 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 – although this depends on the details of the CFD solver and test case. 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.

Despite these lax convergence tolerances, previous work [7] and other literature [8, 9] has shown that repeatedly finding a solution to the linearized pressure equation is a severe bottleneck to the strong scalability of CFD codes. The pressure equation is a stiff, elliptic, Poisson equation which contains many low-frequency error components [10]. The other equations, such as for momentum, are not elliptic and do not usually contain low-frequency errors; the high-frequency errors can be smoothed quickly using less complex smoothers such as Jacobi or Gauss-Seidel methods.

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-frequency 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 [11]), or improved bi-conjugate gradient methods (IBCGS [12]), provide only minor improvements for parallel CFD [7].

Multigrid methods, such as ML [13], are also popular due to their ability to reduce low-frequency errors in a constant number of iterations – regardless of mesh size. The main bottleneck to scalability of multigrid methods is implicit global synchronization, mostly triggered by the Jacobi or Gauss-Seidel smoothing steps on coarser grids. This synchronization occurs because of neighbour-to-neighbour communications, required to communicate ghost cells (or halo data) across processor boundaries in each smoother iteration. 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 smoothing-iteration out-of-sync (a process may start iteration k when its neighbour is still on $k-1$, but it cannot advance any further without waiting for communications). 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 where the cells-per-core ratio is low, and the ratio of communications to computations is high. On the coarsest grids of a multigrid solver the cells-per-core ratio is naturally very low, and implicit synchronization of the coarse-grid smoothers limits scalability of multigrid methods [7].

In order to remove this implicit synchronization from Jacobi-like solvers or smoothers, one can use the concept of 'Chaotic Relaxations' [14] – 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. [15]; but this theory can be leveraged further. Here,

[1] www.refresco.org

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 scalability bottleneck.

In the following section, the theory of chaotic relaxation is introduced and the implementation of this chaotic solver is presented. The solver is tested, via ReFRESCO, on two test cases. 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 then 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.

Although ReFRESCO is used for this testing, only the computational costs and scalability of the pressure-solver is presented throughout. The results should be largely transferable to other scientific models which generate similar elliptic systems. To that end, the chaotic methods discussed in this paper have been written as a standalone library, *Chaos*[2]. The *Chaos* library is available under an open-source, permissive 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. Bindings are also created for Python and Ruby automatically, using SWIG [16]. *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 tool-kits, such as PETSc [17] and Trilinos [18].

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 superscript 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 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. In a parallel solver, some of the x_j variables may not be available in local memory, so neighbour-to-neighbour ('local') MPI communications are required in each iteration.

Numerically, improvements to this scheme can be made by using values from the current iteration (x_j^k, rather than x_j^{k-1}) as they are computed. This is the Gauss-Seidel method, but it is rarely used in its purest form, due to the strictly sequential order of operations [19]. Jacobi and Gauss-Seidel methods are *stationary* methods, so-called because their formulation does not change between iterations. Additionally, they both have an implicit synchronization point, because no two processes can become more than a single iteration out of sync.

In 1969, Chazan and Miranker [14] proposed the theory of *Chaotic Relaxation*, proving convergence when relaxations are performed completely out of sync and in any order. The concept of a global iteration number does not apply, since each equation in the system may be relaxed a different number of times; thus k is now the local iteration number for each equation. Each relaxation uses the values of $\mathbf{x}$ from the latest iteration that is available – this could be several iterations behind the local relaxation iteration ($s = 1, \ldots, k$), or

[2] https://bitbucket.org/jamesnhawkes/chaos

even ahead of it ($s < 0$) if the neighbouring equations have been iterated more frequently:

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

Since the order in which the N equations are relaxed is completely arbitrary it is possible to create a method where processes never need to wait for each other at the end of an iteration. Although communication must still occur, it can be 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. Furthermore, because s is not pre-determined, chaotic relaxations are inherently non-deterministic; this prevents perfect reproducibility of the solution process which may be a limiting factor for some applications.

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.

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 [20] went on to prove that $\rho(|\mathbf{M}|) < 1$ is a necessary condition for convergence for any $s_{max} \leq k$. Bahi [21] further showed that $\rho(|\mathbf{M}|) = 1$ is also convergent, where $\mathbf{M}$ is singular and s_{max} is bounded. Previous work has shown that $\rho(|\mathbf{M}|) \leq 1$ is satisfied by the elliptic equations arising from CFD [10]. A sufficient condition, that $\mathbf{M}$ is diagonally dominant, also guarantees convergence.

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. [15] 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 5th 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 solver, 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.

This 'chaotic solver' is similar in design to the 'racy' Jacobi solver described in Bethune et al. [22] which was shown to be up to 33% faster than synchronous Jacobi in some cases. Bethune et al. found that OpenMP provided a good parallel framework for the racy solver, and suggested that a hybrid-parallel method, as used in *Chaos*, may be ideal. Bethune et al. did not investigate strong scalability, where the increasing ratio of communications to computations could reveal the true advantages of chaotic solvers.

The following sections will describe the implementation and testing of the *Chaos* chaotic solver.

2.1. Implementation

Consider the $N \times N$ 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 $\mathbf{x_B}$.

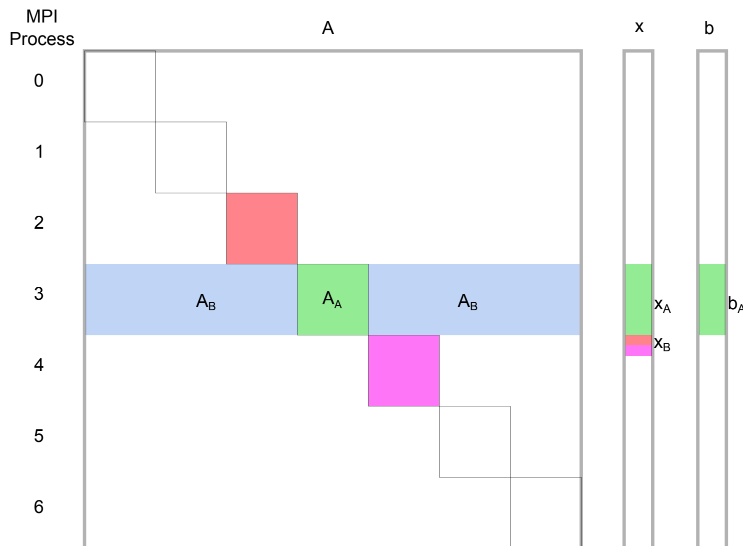

Figure 1: An illustration of the MPI parallel partitioning scheme for the matrix $\mathbf{A}$, and vectors $\mathbf{x}$ and $\mathbf{b}$. Note that the column indices of $\mathbf{A_B}$ are altered to point to the correct location in $\mathbf{x_B}$, the vector-scatter buffer, so that the buffer can be stored as a dense vector.

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 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* (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 [23].

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 3), 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 communication threads will be synchronized. This is not a major concern, but 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 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 necessary 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 at this point.

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 [24], Star-CCM+ [25] or Ansys Fluent [26]. ReFRESCO uses PETSc [17], which provides Krylov subspace solvers and an interface to ML [13] (via the Trilinos project [18]). 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 [27], 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 disk-less, 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 [4]. 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, structured mesh of $N = 2.68$-million cells (139^3) is used[3]. The simulation mimics an infinite domain with two cyclic boundary conditions. The pressure equation is constrained with Dirichlet boundary conditions on the remaining four boundaries – 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 [28] with a Reynolds number of 4.6×10^6. The mesh is a three-dimensional multi-block structured mesh consisting of $N = 2.67$-million cells. A k-ω, two-equation shear stress transport turbulence model is used [29]. Inflow and outflow boundaries are specified appropriately. The pressure equation is bounded by a Neumann condition on the free surface, 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 basic 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 cores. 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 elliptic pressure equation is recorded.

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[4]. 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.

---

[3]Although the mesh is structured, ReFRESCO is an unstructured solver and treats the meshes as an arbitrary (though well-ordered) unstructured mesh.

[4]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.

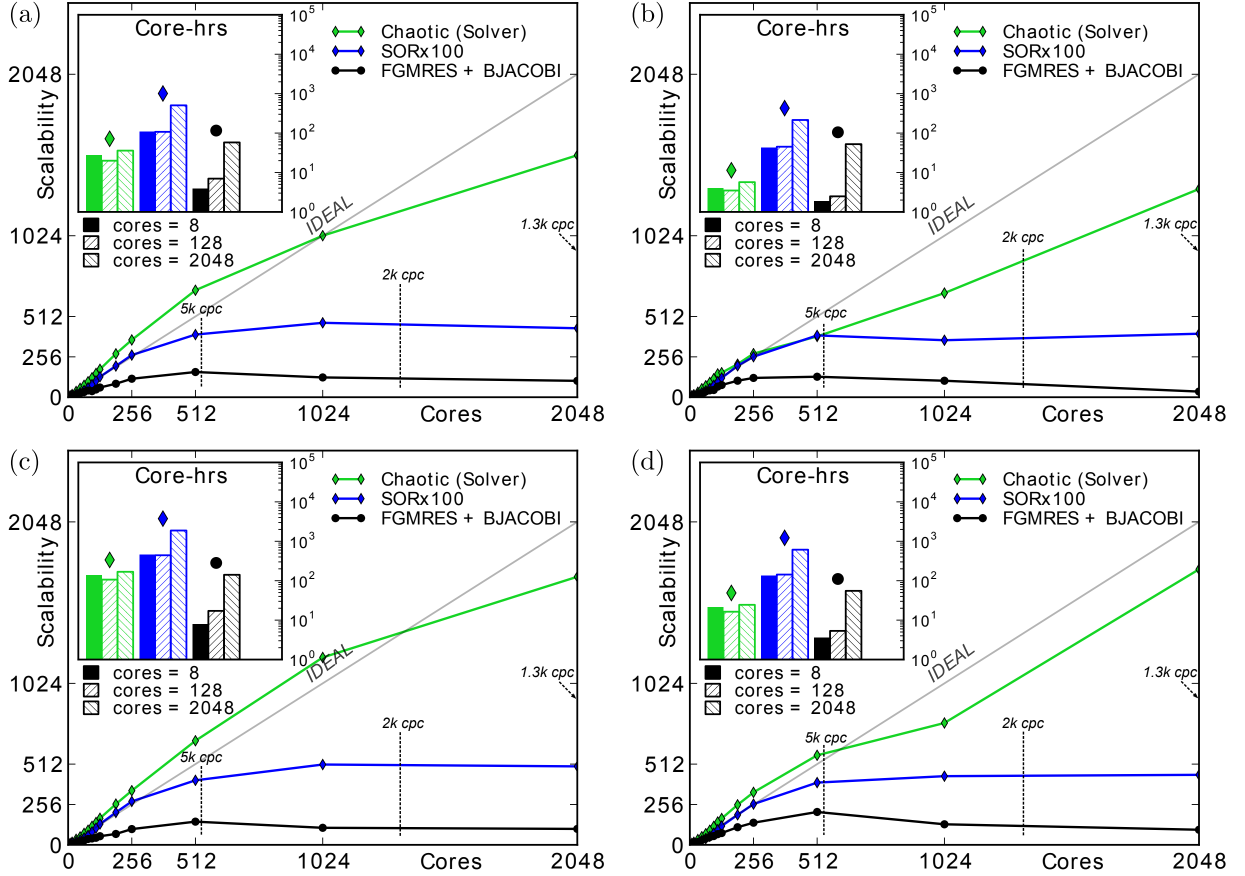

Figure 2: Scalability of the linear equation-system solver, for the pressure equation, for the KVLCC2 (left) and LDCF (right) test cases, at 0.1 (top) and 0.01 (bottom) convergence tolerance. Cells-per-core (cpc) ratio is marked at relevant intervals.

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 $O(N^3)$ [30, §2.2][5]. 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

[5]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 $O(N^3)$.

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 reduce the error to the order of mesh discretization error is constant as N increases, which makes them competitive against Krylov Subspace methods and relevant for real-world applications where N may be large. The work done in each multigrid iteration scales with N, since all of the relevant routines (such as smoothing) have $O(N)$ complexity. Thus, a multigrid solution should converge in $O(N)$-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 [30, §2.4][31].

There are certain traits of the chaotic solver which make it difficult to use 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 is 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.

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 model Poisson equation. Scalability and performance of all the multigrid cycles are then tested against the standard chaotic solver, FGMRES and a state-of-the-art multigrid method (ML) from Trilinos [13, 18].

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, consisting of smoothing on each level and prolongation/restriction to transition between different levels.

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

$$\mathbf{A}_m = \mathbf{R}_m \mathbf{A}_{m-1} \mathbf{P}_m, \quad 1 \leq m \leq m_{max} \tag{4}$$

where $\mathbf{A}_m$ is matrix $\mathbf{A}$ on the mth 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 [32]. 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 less complex 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 [31].

Notay [32] 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 takes care to avoid 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:

$$(\mathbf{A}_m)_{ij} = \sum_{k\in G_i} \sum_{l\in G_j} (\mathbf{A}_{m-1})_{kl} \quad (1 \leq i, j, \leq n_m) \tag{5}$$

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 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 [32]). 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 produce higher-quality coarse grid operators, not least because it would make the number of multigrid levels, and the global size of the coarsest grid, independent of the number of MPI processes.

The second stage of the multigrid algorithm is the actual solution process, in which various different cycles can be used to visit the coarse grids. On each level, a 'correction' to the solution is computed by solving the residual equation on the next-coarsest level. This is known as the additive correction scheme [33]. There are three principal 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.

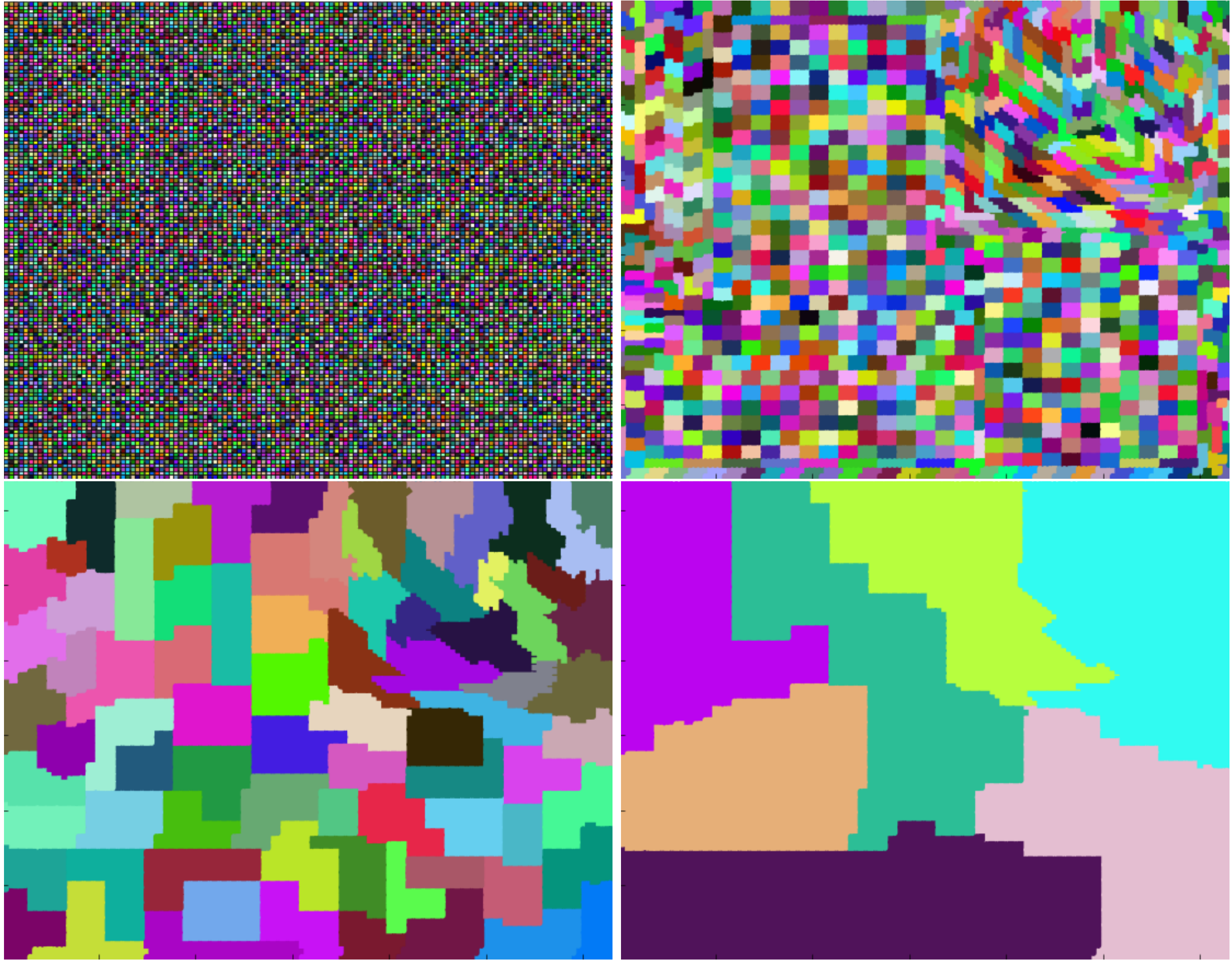

Figure 3: A view of the aggregation process, showing four multigrid levels with $a_{max} = 8$ on a two-dimensional lid-driven cavity flow domain. The fine matrix begins with $128 \times 128 = 16384$ degrees of freedom; and the coarsest matrix has just 7. Aggregates are randomly coloured.

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, W-cycle, F-cycle and sawtooth-cycle are illustrated in figure 4.

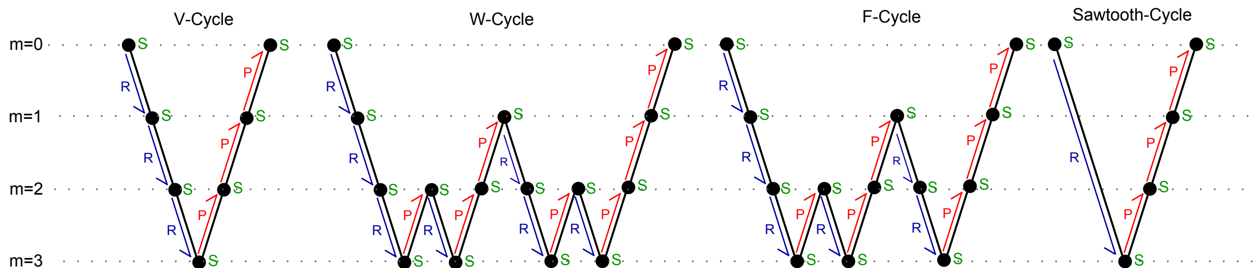

Figure 4: A view of the solution process, showing a single iteration of a four-level multigrid method using the classical recursive V-cycle, W-cycle, F-Cycle and Sawtooth-Cycle. Restriction, Prolongation and Smoothing are colour-coded.

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 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 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. 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 any unsmoothness in the prolongation [30], 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 some relaxations may have used old values ($s > 0$) and therefore this portion of relaxations cannot '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`).

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 (not MPI barrier) 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.

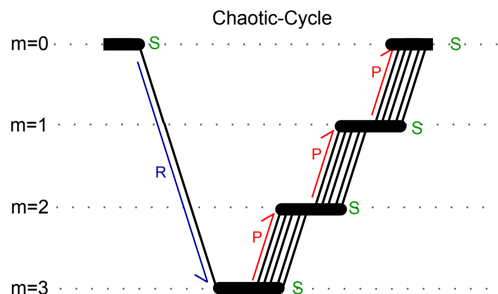

Figure 5: A view of the chaotic-cycle solution process, showing a single iteration of a four-level multigrid method. Restriction, Prolongation and Smoothing are colour-coded. Multiple parallel lines are shown during the prolongation stage to depict the non-deterministic way in which multiple threads (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 (p, q, r) model Poisson equation. A 3D, unit-length, cubic domain of N elements is created (element spacing $h = 1/N^{\frac{1}{3}}$) with Dirichlet boundary conditions. A pre-determined solution $\mathbf{x}_s$ is created for the linear system, by combining 20 sinusoidal frequencies:

$$x_{s,i} = \sum_{f=0}^{19} \sin(2^f \pi pqrh^3) \quad (0 \leq i < N \text{ and } 0 \leq p, q, r < N^{\frac{1}{3}}) \tag{6}$$

$\mathbf{b}$ 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 [31]. Figure 6 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 maximum, 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 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.

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 (non-multigrid) chaotic solver (from section 2), and a state-of-the-art multigrid package (ML [13, 18]). Once again the solution time of the pressure equation is summed

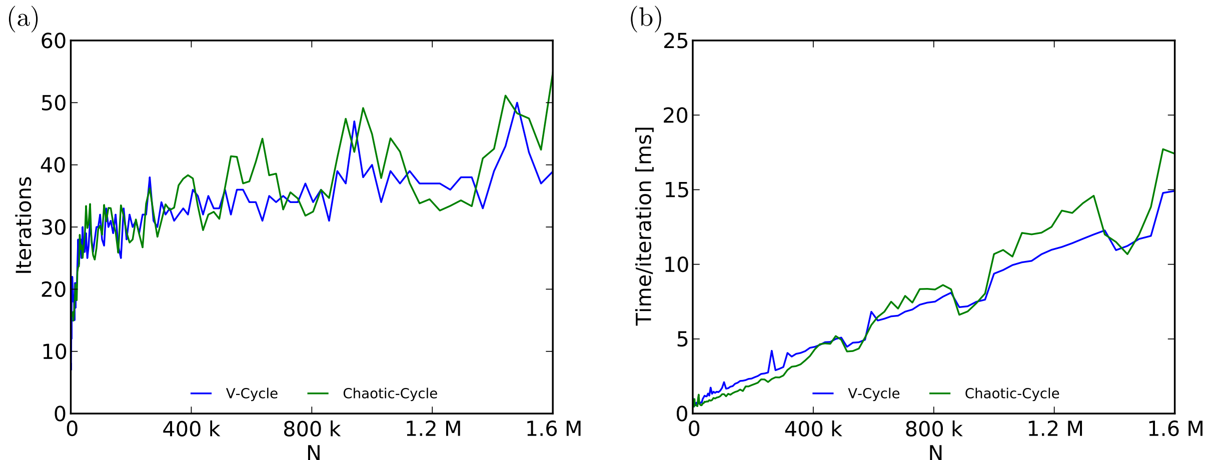

Figure 6: Number of iterations (a) and time-per-iteration (b) to converge the discrete Poisson equation, of size N, to a relative convergence tolerance of 10^{-6}.

Table 1: Absolute core-hours spent solving the linearized pressure equation over 100 non-linear iterations.

Case (tol.)	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

over 100 non-linear iterations 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 non-linear loops, thus the restriction and prolongation matrices do not need recomputing. $\mathbf{A}_m$ must still be refilled on the coarser grids using equation 5. 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 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 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 than the classical V-cycle on $C = 2048$ cores, and 3.7× faster than preconditioned FGMRES. This gap

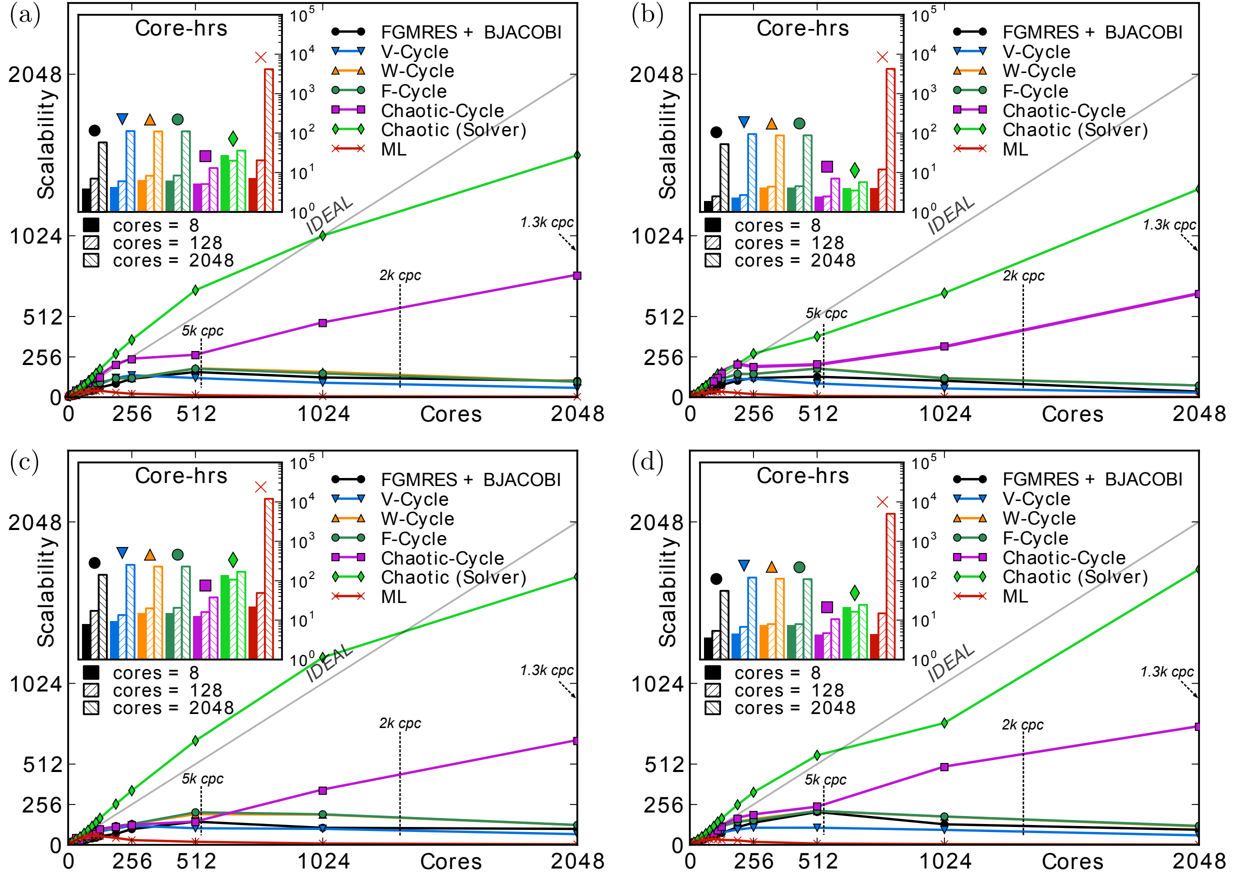

Figure 7: Scalability of the linear equation-system solver, for the pressure equation, for the KVLCC2 (left) and LDCF (right) test cases, at 0.1 (top) and 0.01 (bottom) convergence tolerance. Cells-per-core (cpc) ratio is marked at relevant intervals.

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 does not scale perfectly. Indeed, in one case the plain 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 classical- and chaotic-cycles 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. In the plain chaotic solver, with a residual computed every 100 vector-scatters, residual computations were already the main limitation to overall scalability. Thus, it is suspected that this is the main cause of sub-optimal scaling of the chaotic-cycle multigrid, and is also the only sensible reason that the standard chaotic solver could out-perform the multigrid method (since the chaotic solver is equivalent to a one-level chaotic-cycle). For both the chaotic-cycle multigrid and chaotic solver, the ideal solution would be to apply a threaded/asynchronous 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.

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 from a residual computation, 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 asynchronous 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 and residual checks are slow, it is not clear which 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 basic 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 disadvantaged 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 $7.7\times$ 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 plain 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 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 elliptic 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.

This research is partly funded by the Dutch Ministry of Economic Affairs.

References

[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] S. V. Patankar, Numerical Heat Transfer and Fluid Flow, Hemisphere Publishing Corporation (CRC Press, Taylor & Francis Group), 1980.

[6] C. Klaij, 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) (2013) 830–849.

[7] 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 Accepted for publication.

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

[9] M. Culpo, Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters, Tech. Rep., Partnership for Advanced Computing in Europe, 2011.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] D. Chazan, W. Miranker, Chaotic Relaxation, Linear Algebra and its Applications 2 (2) (1969) 199–222.

[15] 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.

[16] SWIG, `http://www.swig.org`, Acc. 2017.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] I. Bethune, J. M. Bull, N. J. Dingle, N. J. Higham, Performance Analysis of Asynchronous Jacobi's Method Implemented in MPI, SHMEM and OpenMP, The International Journal of High Performance Computing Applications 28 (1) (2014) 97–111.

[23] M. Lange, G. Gorman, M. Weiland, L. Mitchell, J. Southern, Achieving Efficient Strong Scaling with PETSc using Hybrid MPI/OpenMP Optimisation, in: J. M. Kunkel, T. Ludwig, H. W. Meuer (Eds.), Supercomputing, vol. 7905, Springer Berlin Heidelberg, 97–108, URL `http://link.springer.com/chapter/10.1007/978-3-642-38750-0_8`, 2013.

[24] OpenFOAM, OpenFOAM User Guide, 2014.

[25] CD-adapco, STAR-CCM+, http://www.cd-adapco.com, 2014.
[26] ANSYS, FLUENT, http://www.ansys.com, 2011.
[27] VI-HPS, Score-P, v.1.2.3, http://www.vi-hps.org/projects/score-p, Acc. 2013.
[28] 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.
[29] F. Menter, M. Kuntz, R. Langtry, Ten Years of Industrial Experience with the SST Turbulence Model, in: Turbulence, Heat and Mass Transfer 4, Antalya, Turkey, 2003.
[30] P. Wesseling, An Introduction to Multigrid Methods, Pure and Applied Mathematics, John Wiley & Sons Australia, Limited, ISBN 9780471930839, 1992.
[31] 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.
[32] Y. Notay, An Aggregation-Based Algebraic Multigrid Method, Electronic Transactions on Numerical Analysis 37 (6) (2010) 123–146.
[33] B. R. Hutchinson, G. D. Raithby, A Multigrid Method Based on the Additive Correction Strategy, Numerical Heat Transfer 9 (1986) 511–537.