AIAA SciTech
5-9 January 2015, Kissimmee, Florida
AIAA Infotech @ Aerospace

AIAA 2015-1949

Towards High-Performance Optimizations of the Unstructured Open-Source SU2 Suite

Thomas D. Economon[*], Francisco Palacios[†], Juan J. Alonso[‡]
Stanford University, Stanford, CA, 94305, USA

Gaurav Bansal,
Intel Corporation, Hillsboro, OR, 97124, USA

Dheevatsa Mudigere, Anand Deshpande,
Intel Corporation, Bangalore, India

Alexander Heinecke, and Mikhail Smelyanskiy
Intel Corporation, Santa Clara, CA, 95044, USA.

This article presents a snapshot of ongoing efforts to optimize an open-source CFD analysis and design suite, SU2, for high-performance, scalable Reynolds-averaged Navier-Stokes calculations using implicit time integration. We focus on performance optimizations with a particular emphasis on code profiling, the opportunities for parallelism of the software components, and finding highly-scalable algorithms. Consequently, the resulting code modifications are geared toward achieving coarse- and fine-grained parallelism for edge-based, finite volume CFD solvers, making efficient use of memory within a heavily object-oriented solver, and choosing appropriate algorithms for maximizing parallelism, especially when solving the linear systems arising from implicit time integration of the governing equations. All lessons learned are reported for the benefit of both the users and developers of SU2, as well as the larger CFD community for pursuing performance improvements in similar analysis and design software packages.

I. Introduction

The solution of Partial Differential Equations (PDEs) is the basis for predictive simulations in Computational Fluid Dynamics (CFD) that analyze of a wide range of problems, including turbulence, acoustics, structures, materials, heat transfer, vortical flows, and combustion. CFD simulations run the gamut of computational expense, from simple, single-processor jobs to highly-complex computations distributed over millions of processors. Improving the performance of these simulations will allow for more accurate predictions and enable solution methodologies that are currently prohibitively expensive.

The fundamental design challenges faced by conventional single-processor architectures over the past decade compelled the industry to shift in the direction of increased parallelism with multi- and many-core architectures. Practitioners are now forced to look at scaling their workloads not only across distributed memories, but also across large numbers of cores in modern, massively-threaded, shared-memory compute nodes.

While achieving high-performance on modern hardware has become increasingly difficult, we believe that not every engineer needs to become an expert on the diversity of architectures populating the market today.

[*]Postdoctoral Scholar, Department of Aeronautics & Astronautics, AIAA Senior Member.
[†]Engineering Research Associate, Department of Aeronautics & Astronautics, AIAA Senior Member.
[‡]Associate Professor, Department of Aeronautics & Astronautics, AIAA Associate Fellow.

For over three years, we have developed and supported an open-source project, the SU2 software suite;[1,2] an open-source collection of software tools written in C++ for performing CFD analysis and design. It is built specifically for the analysis of PDEs and PDE-constrained optimization on unstructured meshes with state of the art numerical methods, and it is particularly well suited for aerodynamic shape design. The initial applications were mostly in high-speed aerodynamics, but through the initiative of users and developers around the world, SU2 is now being used to analyze and design flows in incompressible settings (mechanical and industrial engineering), renewable energy applications (wind turbines and solar collectors), naval engineering (free surface flows), and even chemical engineering, to name a few. While the framework is general and meant to be extensible to arbitrary sets of governing equations for solving multi-physics problems, the core of the suite is a Reynolds-averaged Navier-Stokes (RANS) solver capable of simulating the compressible, turbulent flows that are characteristic of typical problems in aerospace engineering, and this will be the focus of the present work.

An unstructured-grid flow solver comprises of a diverse range of kernels with varying compute and memory requirements, irregular data accesses, as well as variable and limited amount of instruction-, vector- and thread-level parallelism, which makes achieving high parallel efficiency a very challenging task. In depth performance studies for such large-scale unstructured grid applications are still relatively limited and a focus of active research.[3,4,5] A. Duffy *et al.*[6] evaluated leveraging fine-grained parallelism with early GPUs for the NASA FUN3D code, specifically accelerating only the point implicit solver of FUN3D on GPUs. D. Mudigere *et al.*[7] have done a detailed exploration of the shared memory optimizations for an unstructured Euler code benchmark (PETSc-FUN3D) on modern parallel systems and demonstrated significant performance benefits.

As an open-source package, SU2 is uniquely positioned to serve as an example to computational scientists around the world on how one can achieve high-performance and scalability on advanced hardware architectures. The open-source platform can be leveraged as a testbed for various code optimization strategies and studies on the implications of algorithmic choices. Furthermore, its open-source nature allows for rapid and effective technology transfer. In particular, it should be noted that users and developers in the SU2 community are already benefitting from a number of improvements in the codebase that were motivated directly by this research and were implemented and released into the SU2 suite.

Therefore, the overall goal of the present research is to revisit and optimize an established, open-source CFD analysis and design suite, SU2, from the ground up for execution on modern, highly-parallel (multi- and many-core) architectures. We will place a particular emphasis on parallelism (both fine- and coarse-grained), vectorization, efficient memory usage, and identification of the best-suited algorithms for modern hardware. This article represents a snapshot of our ongoing efforts to optimize SU2 for massively parallel simulations in several key areas:

1. Code profiling and understanding current bottlenecks.
2. Implementation of coarse and fine grain parallelism approaches (SPMD and SIMD) and efficient memory usage.
3. Understanding the implications of algorithmic choices on parallelism, especially for solving the linear systems arising from implicit time discretizations.

The work presented in each of the areas above represents the early progress in a multi-year collaboration focused on the optimization of the SU2 platform. All lessons learned will be reported for the benefit of both the users and developers of SU2, as well as those in the larger CFD community that might be pursuing similar performance improvements. As this is only a snapshot of current progress, we intend for the main contribution of this article to be the explanation of our overall strategy for assessing the current state of the code and the subsequent choices for performance improvement. Future work will focus on the performance of the code in massively parallel settings after implementing a full suite of code optimizations.

The paper is organized as follows. Section II briefly describes the software suite to provide the necessary background for the subsequent performance optimization strategies. Section III describes our basic approach for assessing the performance of SU2 and how early profiling and scalability studies have motivated our code optimization strategy. Sections IV and V contain detailed results and the key conclusions of our work in two areas: code modifications for improving single-node performance (shared memory performance, memory use, and vectorization) and assessing the suitability of a number of available linear solvers for representative RANS calculations (preconditioned Krylov-based methods, classical iterative methods, and multigrid methods). Finally, Section VI summarizes our key conclusions.

II. Code Overview

This section briefly introduces the governing equations and selected numerical methods in order to motivate the chosen optimization strategies for the solver. However, it is important to note that both the software architecture and the choice of algorithms play large roles in parallel performance. Therefore, we will consider modifications of the source code as well as algorithmic changes.

A. Governing Equations

SU2 has been designed to solve PDE-based problems defined on a domain $\Omega \subset \mathbb{R}^3$. In particular, the PDE system resulting from physical modeling of the problem is cast in the following structure:

$$\frac{\partial U}{\partial t} + \nabla \cdot \vec{F}^c - \nabla \cdot (\mu^{vk} \vec{F}^{vk}) = Q \quad \text{in } \Omega,\, t > 0 \tag{1}$$

with appropriate boundary and temporal conditions that will be problem-dependent. In this general framework, U represents the vector of state variables, $\vec{F}^c(U)$ are the convective fluxes, $\vec{F}^{vk}(U)$ are the viscous fluxes, and $Q(U)$ is a generic source term.

We are concerned with compressible, turbulent fluid flows governed by the Reynolds-averaged Navier-Stokes equations. Consider the equations in the domain Ω with a disconnected boundary that is divided into a far-field component Γ_∞ and an adiabatic wall boundary S as seen in Fig. 1. For instance, the surface S could represent the outer mold line of an aerodynamic body. These conservation equations along with the source term Q can be expressed in differential form as

$$\begin{cases} \mathcal{R}(U) = \frac{\partial U}{\partial t} + \nabla \cdot \vec{F}^c - \nabla \cdot (\mu^{vk} \vec{F}^{vk}) - Q = 0 & \text{in } \Omega, \qquad t > 0 \\ \vec{v} = \vec{0} & \text{on } S, \\ \partial_n T = 0 & \text{on } S, \\ (W)_+ = W_\infty & \text{on } \Gamma_\infty, \end{cases} \tag{2}$$

where the conservative variables are given by $U = \{\rho, \rho\vec{v}, \rho E\}^\mathsf{T}$, and the convective fluxes, viscous fluxes, and source term are

$$\vec{F}^c = \left\{ \begin{array}{c} \rho\vec{v} \\ \rho\vec{v} \otimes \vec{v} + \bar{\bar{I}} p \\ \rho E \vec{v} + p\vec{v} \end{array} \right\}, \quad \vec{F}^{v1} = \left\{ \begin{array}{c} \cdot \\ \bar{\bar{\tau}} \\ \bar{\bar{\tau}} \cdot \vec{v} \end{array} \right\}, \quad \vec{F}^{v2} = \left\{ \begin{array}{c} \cdot \\ \cdot \\ c_p \nabla T \end{array} \right\}, \quad Q = \left\{ \begin{array}{c} q_\rho \\ \vec{q}_{\rho\vec{v}} \\ q_{\rho E} \end{array} \right\}, \tag{3}$$

where ρ is the fluid density, $\vec{v} = \{v_1, v_2, v_3\}^\mathsf{T} \in \mathbb{R}^3$ is the flow speed in a Cartesian system of reference, E is the total energy per unit mass, p is the static pressure, c_p is the specific heat at constant pressure, T is the temperature, and the viscous stress tensor can be written in vector notation as

$$\bar{\bar{\tau}} = \nabla\vec{v} + \nabla\vec{v}^\mathsf{T} - \frac{2}{3}\bar{\bar{I}}(\nabla \cdot \vec{v}). \tag{4}$$

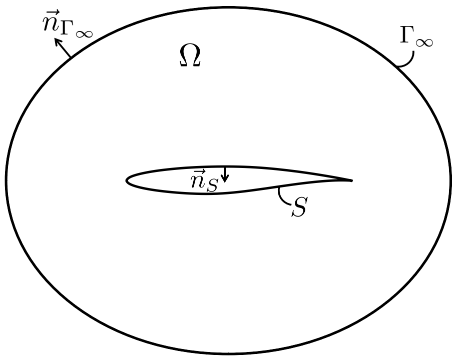


Figure 1. Notional schematic of the flow domain, Ω, the boundaries, Γ_∞ and S, as well as the definition of the surface normals.

The second line of Eqn. (2) represents the no-slip condition at a solid wall, the third line represents an adiabatic condition at the wall, and the final line represents a characteristic-based boundary condition at the far-field[8] with W representing the characteristic variables. In addition to the boundary conditions given in Eqn. (2), the RANS solver in SU2 currently supports a variety of boundary conditions that make the solver suitable for both internal and external aerodynamic calculations.

Assuming a perfect gas with a ratio of specific heats, γ, and gas constant, R, one can determine the pressure from $p = (\gamma - 1)\rho\,[E - 0.5(\vec{v} \cdot \vec{v})]$, the temperature is given by $T = p/(\rho R)$, and $c_p = \gamma R/(\gamma - 1)$. In accord with the standard approach to turbulence modeling based upon the

Boussinesq hypothesis,[9] which states that the effect of turbulence can be represented as an increased viscosity, the total viscosity is divided into a laminar, μ_{dyn}, and a turbulent, μ_{tur}, component. In order to close the system of equations, the dynamic viscosity, μ_{dyn}, is assumed to satisfy Sutherland's law[10] (function of temperature alone), the turbulent viscosity μ_{tur} is computed via a turbulence model, and

$$\mu^{v1} = \mu_{dyn} + \mu_{tur}, \quad \mu^{v2} = \frac{\mu_{dyn}}{Pr_d} + \frac{\mu_{tur}}{Pr_t}, \tag{5}$$

where Pr_d and Pr_t are the dynamic and turbulent Prandtl numbers, respectively.

The turbulent viscosity, μ_{tur}, is obtained from a suitable turbulence model involving the flow state and a set of new variables. The Spalart-Allmaras (S-A)[11] model is one of the most common and widely used turbulence models for the analysis and design of engineering applications affected by turbulent flows, and it will be used for any turbulent numerical experiments in this article. This model requires the solution of an additional scalar PDE sharing the form of Eqn. (1).

B. Numerical Implementation

The following sections contain a brief overview of the numerical implementation strategies for solving PDEs in SU2. Following the method of lines, the governing equations are discretized in space and time separately. This decoupling of space and time allows for the selection of different types of schemes for the spatial and temporal integration. In this article, spatial integration is performed using the finite volume method (FVM), while integration in time is achieved through implicit discretizations that enable large time steps to alleviate stiffness in the equations when marching to a steady solution.

1. *Spatial Integration via the Finite Volume Method*

Partial Differential Equations (PDEs) in SU2 are discretized using a finite volume method[12, 8, 13, 14, 15, 16, 17, 18, 19] with a standard edge-based structure on a dual grid with control volumes constructed using a median-dual, vertex-based scheme. Median-dual control volumes are formed by connecting the centroids, face, and edge-midpoints of all cells sharing the particular node. After integrating the governing equations over a control volume and applying the divergence theorem, the semi-discretized, integral form of a typical PDE (such as the RANS equations above) is given by,

$$\int_{\Omega_i} \frac{\partial U}{\partial t}\, d\Omega + \sum_{j \in \mathcal{N}(i)} (\tilde{F}^c_{ij} + \tilde{F}^{vk}_{ij}) \Delta S_{ij} - Q|\Omega_i| = \int_{\Omega_i} \frac{\partial U}{\partial t}\, d\Omega + R_i(U) = 0, \tag{6}$$

where U is the vector of state variables and $R_i(U)$ is the numerical residual representing the integration of all spatial terms at node i. $\tilde{F}^c_{ij}$ and $\tilde{F}^{vk}_{ij}$ are the projected numerical approximations of the convective and viscous fluxes, respectively, and Q is a source term. ΔS_{ij} is the area of the face associated with the edge ij, $|\Omega_i|$ is the volume of the dual control volume, and $\mathcal{N}(i)$ is the set of neighboring nodes to node i.

The convective and viscous fluxes are evaluated at the midpoint of an edge. The numerical solver loops through all of the edges in the primal mesh in order to calculate these fluxes and then integrates them to evaluate the residual $R_i(U)$ at every node in the numerical grid. The convective fluxes can be discretized using centered or upwind schemes in SU2. A number of numerical schemes have been implemented (JST,[20] Roe,[21] AUSM,[22] HLLC,[19] Roe-Turkel,[23] to name a few), and the code architecture allows for the rapid implementation of new schemes. Slope limiting is applied within upwind schemes in order to preserve monotonicity in the solution by limiting the gradients during higher-order reconstruction (second-order with the MUSCL approach). The slope limiters of Barth and Jesperson[24] and Venkatakrishnan[25] are popular limiter options on unstructured meshes.

Since it will be used below to demonstrate code optimization strategies, we give here a brief description of the JST numerical method for background purposes. The JST scheme approximates the convective flux using a central difference with a blend of two types of artificial dissipation to maintain numerical stability by preventing even-odd decoupling of the solution at adjacent nodes. The artificial dissipation terms are computed using the differences in the undivided Laplacians (higher-order dissipation) of connecting nodes and the difference in the conserved variables (lower-order dissipation) on the connecting nodes. The two levels of dissipation are blended based on a pressure switch for triggering lower-order dissipation in the vicinity of shock waves. The result is a second-order scheme in space. The final expression for the numerical flux using the JST method on unstructured meshes is:[26, 27]

$$\tilde{F}_{ij}^c = \tilde{F}^c(U_i, U_j) = \vec{F}^c\left(\frac{U_i + U_j}{2}\right) \cdot \vec{n}_{ij} - d_{ij}. \tag{7}$$

The artificial dissipation d_{ij} along the edge connecting nodes i and j can be expressed as

$$d_{ij} = \left(\varepsilon_{ij}^{(2)}(U_j - U_i) - \varepsilon_{ij}^{(4)}(\nabla^2 U_j - \nabla^2 U_i)\right)\varphi_{ij}\lambda_{ij}, \tag{8}$$

where the undivided Laplacians $\nabla^2 U$, local spectral radius, stretching in the grid and pressure switches are computed as

$$\begin{aligned}
\nabla^2 U_i &= \sum_{k\in\mathcal{N}(i)} (U_k - U_i), & (9)\\
\lambda_{ij} &= \left(|(\vec{u}_{ij} - \vec{u}_{\Omega_{ij}}) \cdot \vec{n}_{ij}| + c_{ij}\right)\Delta S, \quad \lambda_i = \sum_{k\in\mathcal{N}(i)} \lambda_{ik}, & (10)\\
\varphi_{ij} &= 4\frac{\varphi_i\varphi_j}{\varphi_i + \varphi_j}, \quad \varphi_i = \left(\frac{\lambda_i}{4\lambda_{ij}}\right)^{\alpha}, & (11)\\
\varepsilon_{ij}^{(2)} &= \kappa^{(2)} s_2 \left(\left|\sum_{k\in\mathcal{N}(i)} (p_k - p_i)\right| \Big/ \sum_{k\in\mathcal{N}(i)} (p_k + p_i)\right), & (12)\\
\varepsilon_{ij}^{(4)} &= s_4 \max\left(0, \kappa^{(4)} - \varepsilon_{ij}^{(2)}\right), & (13)
\end{aligned}$$

where $\mathcal{N}(i)$ represents the set of neighboring points to node i, p_i is the pressure at node i, s_2 and s_4 are stretching parameters, α is typically set to 0.3, and $\kappa^{(2)}$ and $\kappa^{(4)}$ are adjustable parameters (typical values on unstructured meshes are $\kappa^{(2)} = 0.5$ and $\kappa^{(4)} = 0.02$).

In this work, the convective term for the scalar variable in the S-A turbulence model is discretized using an upwind scheme. Typically, a first-order scheme is chosen, but the turbulence variable can also take advantage of a MUSCL approach and slope limiters in order to obtain second-order accuracy.

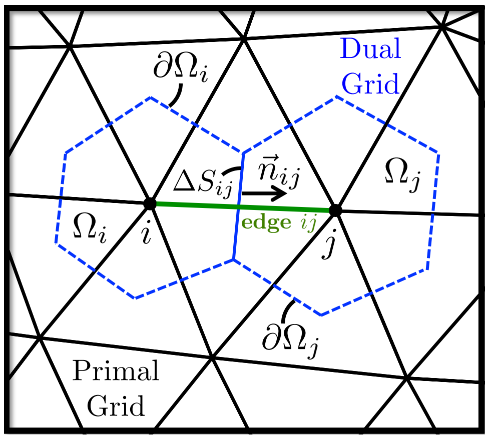


Figure 2. Dual mesh control volumes surrounding two nodes, i and j, in the domain interior.

In order to evaluate the viscous fluxes using a finite volume method, flow quantities and their first derivatives are required at the faces of the control volumes. The gradients of the flow variables are calculated using either a Green-Gauss or weighted least-squares method at all grid nodes and then averaged to obtain the flow variable gradients at the cell faces. Source terms are approximated at each node using piecewise constant reconstruction within each of the dual control volumes.

The FVM with a median-dual, cell-vertex scheme, as described above, is amenable to an edge-based data structure, meaning that mesh data (nodal coordinates, face areas, local normals, etc.) are stored on an edge-by-edge basis. The edge-based structure offers convenience in the implementation, as a single loop over the edges in the mesh allows for the numerical fluxes for each node to be computed. In SU2, a number of classes exist for encapsulating the various components of the geometry, such as nodes and edges.

In practice, the numerical residual $R_i(U)$ (we will see that this eventually becomes the right-hand side for our implicit solve) is evaluated with each nonlinear iteration using a sequence of loops over the edges and nodes:

1. Loop over all of the edges in the primal mesh in order to calculate the convective and viscous fluxes.
2. Loop over all of the nodes in the primal mesh and compute source terms in each dual control volume given the current state.

3. Loop over all of the boundary nodes in the primal mesh in order to impose boundary conditions.

This series of steps results in a value of $R_i(U)$ at each node at a single instance in time, which can then be substituted into Eqn. (6) and integrated forward in time to either arrive at a steady state or a time-accurate solution for the state vector U. Additional loops over the edges and grid nodes may appear depending upon the chosen numerical methods (slope limiters, for example) or for updating the solution at each node after a nonlinear iteration of the fluid equations, for instance.

2. Time Integration

We now consider the techniques for time-marching the coupled system of ordinary differential equations for the flow problem represented by Eqn. (6), which is repeated here:

$$\frac{d}{dt}\int_{\Omega_i(t)} U \, d\Omega + R_i(U) = 0. \tag{14}$$

By discretizing the time derivative term, one obtains a fully-discrete finite volume form of the governing equations. The choice of time-marching method depends on whether a steady state or a time-accurate solution is desired. In both cases, explicit and implicit methods are available. For simplicity, Eqn. (14) can be rewritten as

$$\frac{d}{dt}\left(|\Omega_i| U_i\right) + R_i(U) = 0, \tag{15}$$

where $|\Omega_i| = \int_{\Omega_i(t)} d\Omega$.

For particularly stiff problems (often the case with the RANS equations on stretched grids with high aspect ratio cells), the small time step requirement for explicit schemes may become prohibitive, and implicit methods can be used to improve convergence due to their increased numerical stability. Perhaps the most common implicit method for steady flows is the backward Euler scheme, where the residual is evaluated using the solution state at the new time level U^{n+1}. Applying this to Eqn. (15), one has

$$|\Omega_i|\frac{\Delta U_i}{\Delta t_i} = -R_i(U^{n+1}), \tag{16}$$

where time level n corresponds to the known solution in its current state, while time level $n+1$ represents the new solution state that is being sought after advancing one time step Δt where $\Delta t = t^{n+1} - t^n$ and $\Delta U_i = U_i^{n+1} - U_i^n$. However, the residuals at time level $n+1$ are now a function of the unknown solution state U^{n+1} and can not be directly computed. To remedy this, a first-order linearization about time level n can be performed:

$$\begin{aligned} R_i(U^{n+1}) &= R_i(U^n) + \frac{\partial R_i(U^n)}{\partial t}\Delta t_i^n + \mathcal{O}(\Delta t^2) \\ &= R_i(U^n) + \sum_{j \in \mathcal{N}(i)} \frac{\partial R_i(U^n)}{\partial U_j}\Delta U_j^n + \mathcal{O}(\Delta t^2). \end{aligned} \tag{17}$$

Introducing Eqn. (17) into Eqn. (16), we find that the following linear system should be solved to find the solution update (ΔU_i^n):

$$\left(\frac{|\Omega_i|}{\Delta t_i^n}\delta_{ij} + \frac{\partial R_i(U^n)}{\partial U_j}\right) \cdot \Delta U_j^n = -R_i(U^n), \tag{18}$$

where if a flux $\tilde{F}_{ij}$ has a stencil of points $\{i, j\}$, then contributions are made to the Jacobian at four points, or

$$\frac{\partial R}{\partial U} := \frac{\partial R}{\partial U} + \begin{bmatrix} \ddots & & & & \\ & \frac{\partial \tilde{F}_{ij}}{\partial U_i} & \cdots & \frac{\partial \tilde{F}_{ij}}{\partial U_j} & \\ & \vdots & \ddots & \vdots & \\ & -\frac{\partial \tilde{F}_{ij}}{\partial U_i} & \cdots & -\frac{\partial \tilde{F}_{ij}}{\partial U_j} & \\ & & & & \ddots \end{bmatrix}. \tag{19}$$

Note that, while implicit schemes offer more stability, the use of approximate Jacobians (first-order) can impose limits on the allowable time step, especially at the beginning of the solution process when we are not near the converged solution. However, implicit methods enable the use of higher CFL conditions than with explicit methods, which translate to the specific values of Δt_i^n that are used to relax the problem.

For steady problems, a constant time step for all cells is not required, and a local time-stepping technique can be used to accelerate convergence to a steady state. Local time-stepping allows each cell in the mesh to advance the solution at a local time step that can be calculated from an estimation of the spectral radii at every node i according to

$$\Delta t_i = N_{CFL} \min \left(\frac{|\Omega_i|}{\lambda_i^{conv}}, \frac{|\Omega_i|}{\lambda_i^{visc}} \right), \tag{20}$$

where N_{CFL} is the Courant-Friedrichs-Lewy (CFL) number and λ_i^{conv} is the integrated convective spectral radius[28] computed as

$$\lambda_i^{conv} = \sum_{j \in \mathcal{N}(i)} \left(|\vec{u}_{ij} \cdot \vec{n}_{ij}| + c_{ij} \right) \Delta S, \tag{21}$$

where $\vec{u}_{ij} = (\vec{u}_i + \vec{u}_j)/2$ and $c_{ij} = (c_i + c_j)/2$ denote the velocity and the speed of sound at the cell face as an average of the neighhboring nodes, respectively. The viscous spectral radius λ_i^{visc} is computed as

$$\lambda_i^{visc} = \sum_{j \in \mathcal{N}(i)} C \frac{\mu_{ij}}{\rho_{ij}} S_{ij}^2, \tag{22}$$

where C is a constant, μ_{ij} is the sum of the laminar and eddy viscosities in a turbulent calculation and ρ_{ij} is the density evaluated at the midpoint of the edge ij.

3. Linear Solvers for Implicit Integration

With each nonlinear iteration of the RANS solver, the system in Eqn. (18) is smoothed for some number of linear iterations or until reaching a prescribed convergence tolerance. Common linear solver choices for modern solvers include classic iterative methods and preconditioned Krylov methods. Preconditioning is the application of a transformation to the original system that makes it more suitable for numerical solution.[29] In particular, Jacobi, Lower-Upper Symmetric-Gauss-Seidel (LU-SGS), line implicit (Linelet), and Incomplete LU (with no fill in, i.e., ILU(0)) preconditioners have been implemented to improve the convergence rate of the available linear solvers.[30,31] Currently, the following two Krylov subspace methods are available in SU2:

- The Generalized Minimal Residual (GMRES) method.[32]
- The Biconjugate Gradient Stabilized (BiCGSTAB) method.[33]

In this work, we will mostly focus on the GMRES method.

More recently, a geometric linear multigrid algorithm has also been implemented in SU2 for solving the linear systems arising from an implicit discretization. The geometric multigrid method generates effective convergence at all length scales of a problem by employing a sequence of grids of varying resolution. To illustrate the basic components of linear multigrid, consider solving a linear system on a pair of grids (fine and coarse) of the form

$$A\,\Phi = \mathbf{f}, \tag{23}$$

define the error in the solution to be the difference between the solution Φ and the approximation to the solution $\tilde{\Phi}$, or

$$\mathbf{e} = \Phi - \tilde{\Phi}, \tag{24}$$

where $\mathbf{e}$ is the error vector. We can also define a residual vector, which is a measure of how well the discretized governing equations are being satisfied by our numerical solution procedure, as

$$\mathbf{r} = \mathbf{f} - A\,\tilde{\Phi}, \tag{25}$$

where $\mathbf{r}$ is our residual vector. We can introduce Eqn. (24) into our original system in Eqn. (23) to give

$$A \left(\tilde{\Phi} + \mathbf{e} \right) = \mathbf{f}, \tag{26}$$

and by introducing Eqn. (25), we recover the following expression:

$$A\,\mathbf{e} = \mathbf{r}, \tag{27}$$

which relates the error in the solution to the residual. In practical use with multigrid, Eqn. (27) allows us to compute a measure of the error on coarser mesh levels after transferring the values of the residual from the fine mesh level onto the coarse level (restriction). Computing this measure of error requires the selection of a suitable smoother, which is typically a classical iterative method or a Krylov-based method. After calculating $\mathbf{e}$ on a coarse level to satisfy some desired level of convergence, we can form a correction to the solution on the fine mesh (residual correction) as

$$\Phi = \tilde{\Phi} + \mathbf{e}, \tag{28}$$

after we transfer the error back to the fine mesh from the coarse mesh level (prolongation). Furthermore, we can apply this idea recursively over an entire set of grids of various resolutions to complete an entire multigrid cycle. We will explain more details about the multigrid algorithm below as part of the discussion on nonlinear multigrid.

4. Convergence Acceleration via Nonlinear Multigrid

Nonlinear multigrid is often used as a convergence acceleration technique for solving the fluid equations, regardless of whether an explicit or implicit method is chosen for the time derivative term (i.e., for solving the outer loop of the nonlinear governing equations).

The key idea of the multigrid algorithm is to accelerate the convergence of the numerical solution of a set of equations by computing corrections to the fine-grid solutions on coarse grids and applying this idea recursively.[31,34,35] It is well known that, owing to the nature of most iterative methods/relaxation schemes, high-frequency errors are usually well damped, but low-frequency errors (global error spanning the solution domain) are less damped by the action of iterative methods that have a stencil with a local area of influence.

An agglomeration Full Approximation Storage (FAS) multigrid has been implemented in SU2. The basic methodology is described below. Consider the nonlinear problem $L(w) = f$ defined in a domain Ω, and denote its discretization on a fine grid with spacing h as

$$L_h(u_h) = f_h, \text{ in } \Omega_h, \tag{29}$$

where $L_h(\cdot)$ is a nonlinear discrete operator defined in Ω_h. The starting point is the definition of a suitable smoother (e.g. Jacobi, SGS, GMRES, etc.) and, after a small number of iterations of this method (possibly a single one, instead of fully solving the discrete equation), an approximate solution $\bar{u}_h$ and residual r_h are obtained on the fine grid. The resulting equation in the fine grid can be written as

$$L_h(\bar{u}_h) - f_h = r_h. \tag{30}$$

Subtracting equations (29) and (30) we obtain the following expression to be approximated in a coarse grid:

$$L_h(u_h) - L_h(\bar{u}_h) = -r_h, \tag{31}$$

where the exact solution u_h can be expressed as the approximate solution plus a correction c_h yielding:

$$L_h(\bar{u}_h + c_h) - L_h(\bar{u}_h) = -r_h. \tag{32}$$

Note that no assumptions about the linearity of the operator $L(\cdot)$ (or its discrete version) are made. As we stated before, the objective is to write (32) on a coarse grid of spacing H. In order to do that, two types of restriction operators will be defined: I_h^H, the restriction operator that interpolates the residual from the fine grid h to the coarse grid H (in a conservative way), and $\bar{I}_h^H$, which simply interpolates the fine grid solution onto the coarse grid. Formulating (32) on the coarse level by replacing $L_h(\cdot)$ by $L_H(\cdot)$, $\bar{u}_h$ by $\bar{I}_h^H \bar{u}_h$, and r_h by $I_h^H r_h$, we obtain the FAS equation:

$$L_H(\bar{I}_h^H \bar{u}_h + c_H) - L_H(\bar{I}_h^H \bar{u}_h) = -I_h^H r_h. \tag{33}$$

In this last expression, by definition, the approximate solution on the coarse grid is denoted as $\bar{u}_H := \bar{I}_h^H \bar{u}_h + c_H$, and the residual r_h can be written as $L_h(\bar{u}_h) - f_h$. Finally we obtain the following useful equation on the coarse level:

$$L_H(\bar{u}_H) = L_H(\bar{I}_h^H \bar{u}_h) - I_h^H (L_h(\bar{u}_h) - f_h). \tag{34}$$

This last expression can also be simply written as

$$L_H(\bar{u}_H) = f_H + \tau_h^H, \text{ in } \Omega_H, \tag{35}$$

where the source term on the coarse levels is interpolated $f_H = I_h^H f_h$ (not computed), and a new variable $\tau_h^H = L_H(\bar{I}_h^H \bar{u}_h) - I_h^H (L_h \bar{u}_h)$ is defined as the fine-to-coarse defect or residual correction. Note that without the τ_h^H term the coarse grid equation is the original system represented on the coarse grid.

The next step is to update the fine grid solution. For that purpose the coarse-grid correction c_H (which in principle is smooth because of the application of the smoothing iteration) is interpolated back on to the fine grid using the following formula

$$\bar{u}_h^{new} = \bar{u}_h^{old} + I_H^h (\bar{u}_H^{new} - \bar{I}_h^H \bar{u}_h^{old}), \tag{36}$$

where I_H^h is a prolongation operator that interpolates coarse grid correction to the fine grid. Note that we interpolate the correction and not the coarse-grid solution itself.

In this brief introduction to the method, only two grids have been considered. In real problems, however, the algorithm is applied in a recursive way using different grid level sizes to eliminate the entire spectrum of frequencies of the numerical error. In order to summarize the method, the basic multigrid algorithm is presented in pseudo-code below:

```
Algorithm FAS Multigrid
1.  if $k = 1$
2.     then solve $L_k(u_k) = f_k$ directly
3.  for $l \leftarrow 1$ to $\nu_1$
4.        do Pre-smoothing steps on the fine grid:
5.           $u_k^{(l)} \leftarrow S(u_k^{(l-1)}, f_k)$
6.  Computation of the residual: $r_k \leftarrow f_k - L_k(w_k^{(\nu_1)})$
7.  Restriction of the residual: $r_{k-1} \leftarrow I_k^{k-1} r_k$
8.  $u_{k-1} \leftarrow \bar{I}_k^{k-1} u_k^{(\nu_1)}$
9.  $f_{k-1} \leftarrow r_{k-1} + L_{k-1}(u_{k-1})$
10. Call $\gamma$ times the FAS scheme to solve $L_{k-1}(u_{k-1}) = f_{k-1}$ using a V cycling strategy
11. Coarse-grid correction: $u_k^{new} \leftarrow u_k^{(\nu_1)} + I_{k-1}^k (u_{k-1} - \bar{I}_k^{k-1} u_k^{(\nu_1)})$
12. for $l \leftarrow 1$ to $\nu_1$
13.       do Post-smoothing steps on the fine grid:
14.          $u_k^{(l)} \leftarrow S(u_k^{(l-1)}, f_k)$
```

Because of their structured-grid heritage, multigrid methods have traditionally been developed from a geometric point of view. In this particular implementation, a geometric agglomeration multigrid method has been used. This strategy consists in choosing a seed point (a node in a vertex-based code such as SU2), which initiates a local agglomeration process whereby the neighboring control volumes are agglomerated onto the seed point. The topological fusing for the agglomeration multigrid method is a fundamental component of the algorithm: a number of priorities and restrictions are imposed on the agglomeration process to ensure high quality (maximum number of points, volume, ratio surface/volume, boundary incompatibilities, etc.). The most important advantage of the agglomeration technique is that it is not necessary to physically create independent meshes on the coarse levels: this task can be completely automated.

To avoid confusion, it is important to note the differences between the nonlinear (FAS) multigrid algorithm described here and the linear multigrid algorithm described above for solving a linear system. The nonlinear multigrid algorithm requires that the residual vector and Jacobian matrix of the problem (i.e., the matrix and right-hand side of the linear system) are recomputed on each level before smoothing occurs. In the linear multigrid case, the residual vector and Jacobian matrix are computed once at the beginning of each nonlinear iteration of the equations and held fixed during the smoothing and traversal of the multigrid levels. In SU2, we leverage the same agglomeration techniques, restriction operators, and prolongation operators for both the linear and nonlinear multigrid methods.

III. Approach to Code Optimization

The first step in our approach was to create a new C++ module named SU2_PHI, which contains a stripped down version of the main SU2 suite in a standalone executable. More specifically, this module contains the entire RANS solver found in the SU2_CFD module, including the mesh partitioning and MPI routines needed for parallel execution on distributed memory clusters. The motivation for SU2_PHI was the need for a light-weight environment in which to prototype the various modifications to the code and new algorithms while maintaining all of the necessary physics and the complexity of the full software architecture.

With SU2_PHI, we could then select a number of representative problems from aeronautics and begin the process of improving the performance and scalability of SU2 by performing a detailed code profiling and initial scalability tests. Through these tests, we can appropriately prioritize the routines that should receive attention based on their workload and potential for improved parallelization. In general, additional parallelization of routines will be pursued where possible through typical approaches (decomposition, vectorization, etc.), but ameliorating some bottlenecks in parallelization may require algorithmic changes.

The key challenge during all performance optimizations will be to maintain open-source customizability of the source code while maximizing performance and scalability. Code readability and ease of modification are critical features to our user base (object-oriented C++), and portability and hassle-free installation must be maintained. Traditionally, we have avoided the inclusion of third-party libraries in lieu of optimized, in-house solutions. However, we may need new interfaces to external packages in order to achieve the performance improvements that we seek.

A. Problem Selection

The following representative problems from aeronautics have been selected for our numerical studies:

1. Compressible, turbulent flow around the NACA 0012 airfoil (2D).
2. Transonic, inviscid flow over the ONERA M6 wing (3D).
3. Transonic, turbulent flow around the RAE 2822 airfoil (2D).
4. Transonic, turbulent flow over the NASA Common Research Model (CRM) aircraft configuration (3D).

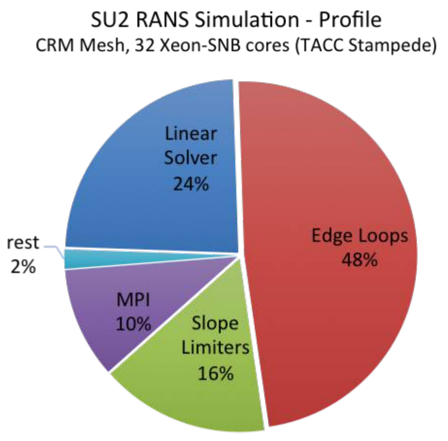
These are some of the most widely used geometries for verification and validation of CFD codes, and these geometries have a wealth of data available from experiment and also other codes for comparison. Another key component in selecting these configurations is the fact that we have several series of grids with varying resolutions for performing numerical experiments at different scales while maintaining sufficient work per core (from single-node calculations up to large-scale, distributed calculations using thousands of MPI ranks).

B. Code Profiling and Initial Scalability Tests

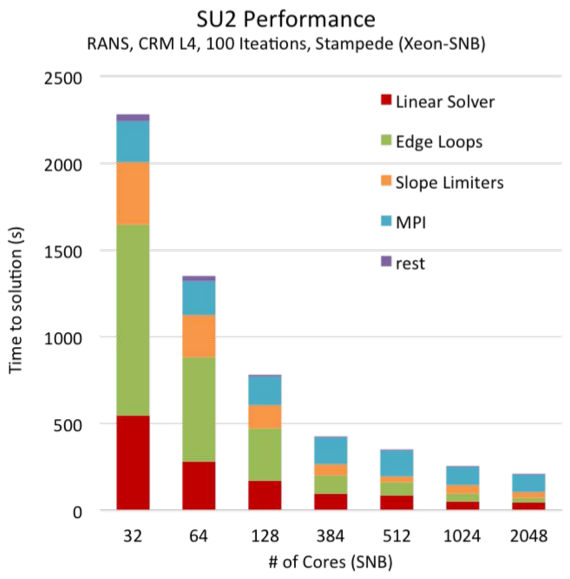
Apart from using other available profiling tools, we have implemented a custom profiling capability within SU2_PHI. This capability allows for the gathering of statistics on routines such as total number of calls, minimum or maximum time for any one call, or the average time of each call to the routine. In addition, the profiling routines are MPI-aware so that information can be gathered for each rank in a calculation independently. Lastly, routines can be tagged with particular identifiers during profiling in order to organize the output so that routines are grouped by their level in the software stack or in a different manner chosen by the user.

Fig. 3 contains initial code profiling and scalability results for the baseline version of SU2_PHI. As expected for an unstructured, edge-based, implicit CFD solver, such as the structure described above for SU2, almost all of the computational work is found in the following:

- Edge loops/point updates (convective and viscous fluxes, time steps, slope limiters, etc.)



(a) Single-node profiling results.



(b) Initial scalability results.

Figure 3. Profiling and initial scalability results for the baseline version of SU2_PHI.

- Linear solvers (approximate factorization, preconditioning and back-substitution - sparse linear algebra)
- Global collectives (vector inner products and norms, marked as MPI in Fig.3)

The majority of the execution time is spent in the edge-based loops (the "physics" of the application) and the linear solver (sparse, narrow-band recurrences). The collectives typically have very little floating point work and involve a logarithmically deep succession of messages to traverse the subdomains of the partition. In the distributed memory context, edge-based loops are bound by the inter-node bandwidth if the latter does not scale with the architecture. Inner products are bound by the inter-node latency and network diameter. However, the single node (shared-memory) challenges are different, and are described in the next section for both edge-based loops and recurrences.

1. *Edge-based Loops*

This comprises the stencil computations over all the edges and the associated vertices. These loops predominantly occur in residual vector (flux) calculation, Jacobian matrix evaluation, and Jacobian-vector products. Typically, these loops have color-wise concurrency and local communication to complete the edges cut by the mesh partitioning for distributed calculations. A typical edge-based loop is depicted in Listing. 1. These edge loops contribute to the majority of the floating-point operations and hence are expected to be bound by available compute on the processor. Being compute-bound, we expect these loops to scale with the increasing compute (nodes), and with our initial scaling studies, we see that these loops have near-linear performance scaling. However, the key challenges in achieving the potential performance is: 1) Extracting thread- and SIMD-level parallelism in the presence of loop-carried dependencies, due to vertices shared by multiple edges, and 2) Exploiting SIMD-level parallelism in the presence of irregular memory accesses.

2. *Linear Solvers*

This consists of sparse linear algebra kernels that are generally characterized by sparse, narrow-band recurrences. These operations have limited parallelism, proportional to the number of independent edges. Also, the compute intensity is quite low, making these operations memory intensive and typically bound by the memory bandwidth. Furthermore, due to the varying amount of parallelism these operations are also plagued by significant amounts of load imbalance when parallelized.

Currently, the block-CSR storage format (BCSR) is used in SU2 for storing the Jacobian matrix with the block size being the number of unknowns per vertex (5×5). Using BCSR has significant benefits, since it allows for coalesced loads (4 cache lines per block), reduces the index computation, and also alleviates the memory bandwidth pressure

IV. Single-node Performance Optimizations

In this section, we will describe the various code changes we make to improve the single-node performance for a given workload. These fine-grained optimizations can be grouped into three major categories:

- Code hybridization to MPI + OpenMP to take advantage of shared-memory resources. This also helps in reducing the overall memory requirement.
- Efficient memory use. This includes three things: Firstly, re-ordering the edges of the unstructured mesh using a Reverse Cuthill-Mckee (RCM) algorithm to reduce the bandwidth of the edge adjacency matrix. Secondly, doing a smart memory allocation for objects such that contiguous blocks of memory can be allocated for them. And lastly, array-of-structures to structures-of-arrays (AOS to SOA) type transformations. All these help in compacting the memory access footprint for any function such that the cache miss rate can be reduced.
- Vectorization. Vectorization is of fundamental importance in getting the performance out of modern computer architectures.

For the results presented in this section, we select an inviscid, transonic ONERA M6 workload together with Runge-Kutta (RK) explicit time-stepping scheme. This forms a building-block for more involved turbulent and implicit time-stepping simulations, and at the same time retains the edge-loops which form the top hotspots for any other workload. Moreover, since this forms the bare-bones of the SU2 solver, it has one of the lowest overall compute intensities (compute flops per byte of memory accessed) and as such it is most difficult to optimize (apart from the complications associated with the linear solver for an implicit time-stepping case). If any other extra modules are added (such as turbulence or finite-rate chemistry) the compute intensity should increase driving the performance higher.

In the next subsections, we give a detailed description of the optimizations mentioned above and present performance results for two different unstructured (tetrahedral) meshes: small mesh (94,493 grid points), and large mesh (818,921 grid points). Performance results for both Intel® Xeon®[a] and Intel® Xeon Phi™[a] architectures are presented.

A. OpenMP (OMP) Approach

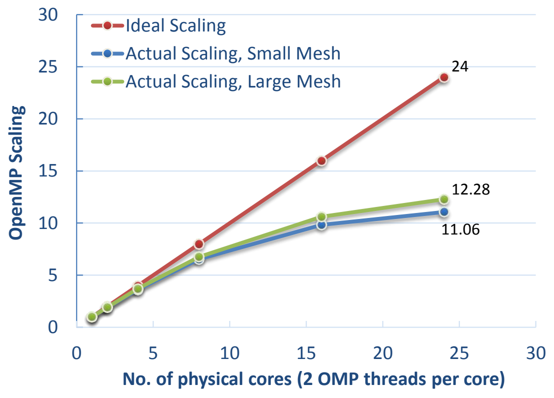
Here we will be discussing our OpenMP approach. OpenMP can be implemented in a couple of ways:

- Loop-level OMP parallel regions. This is easy to implement as one can incrementally add OMP parallelization. Also, this is less error prone due to implicit barriers at the end of the parallel regions. However, this incurs a large fork-join overhead as the number of parallel regions are usually pretty high.
- High-level, functional, OMP approach. This approach involves a single OMP parallel region at a very high-level in the program. This approach looks similar to MPI domain decomposition. Here, the iteration space for edge loops is pre-divided by coloring the edges such that one color belongs to a given OMP thread. This way one can even avoid OMP atomics by pre-assigning the "owner" thread for a vertex which is shared between the edges. One can use OMP dynamic scheduling instead of pre-dividing iteration space if the load-imbalance is substantial. This approach is more difficult to implement and is more error prone because all the synchronizations need to be explicitly managed by the developer. However, this approach offers better performance when implemented carefully.

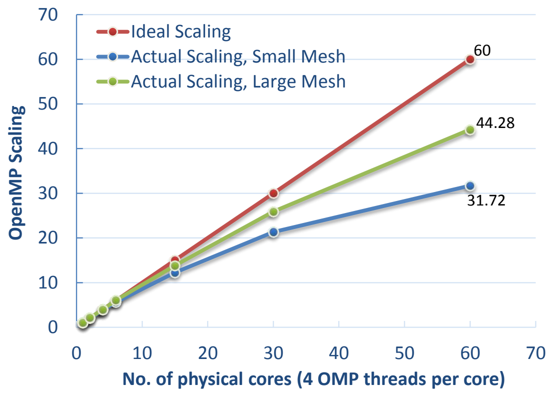
[a]Intel Xeon and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

We implemented both the OMP approaches described above. However, we retained the latter approach as it out-performed the former.

Figs. 4(a) and (b) show the OpenMP strong-scaling for Xeon® and Xeon Phi™, respectively. Note that hyperthreading is enabled for Xeon® such that 2 OMP threads are affinitized (compactly) to a physical core for Xeon®. 4 OMP threads are affinitized (again compactly) to a physical core for Xeon Phi™ to take advantage of the 4 hardware-threads per physical core. This helps hide the latency associated with in-order execution on a Xeon Phi™ core. The results are shown for the two meshes, i.e., small and large. For Xeon® we see that maximum scaling achieved is 11.06x for the small mesh and 12.28x for the large mesh. For Xeon Phi™ the corresponding numbers are 31.72x and 44.28x. The large mesh shows better scaling compared to the small mesh because the effects of OMP load-imbalance reduce as the amount of compute increases. This is even more so for Xeon Phi™ as the number of OMP threads is higher.



(a) Intel® Xeon® results.



(b) Intel® Xeon Phi™ results.

Figure 4. OpenMP strong scaling. As indicated, 2 OMP threads are affinitized to a physical core for Xeon® and 4 OMP threads are affinitized to a physical core for Xeon Phi™. Compact affinity is used in both cases.

Next, we describe in more detail our OpenMP implementation. The top hotspots for the inviscid mean flow workload with explicit R-K time integration consists of essentially the edge-loops related to the convective flux calculation:

1. SetMax_EigenValue
2. SetUndivided_Laplacian
3. SetDissipation_Switch
4. Centered_Residual

Out of these 4 hotspots, the first three compute the various auxiliary parameters required for JST flux residual (see Section II) computation, which gets computed in the fourth function, Centered_Residual. One optimization was to combine the second and third function into one to take advantage of the temporal cache-locality. Since they share some of the memory accesses, it makes sense to bring the data from memory once and compute on it as much as possible. This new function is called Set_UndLapl_DissSwitch.

A typical edge-loop in any of these functions looks like Listing 1. The structure corresponds to a gather-compute-scatter method, in that for each loop iteration, data is gathered from the corresponding vertices of the edge, some compute is performed using data from both vertices, and the results of compute are scattered back to the vertices. The SetMax_EigenValue and Set_UndLapl_DissSwitch are characterized by a low compute intensity. The Centered_Residual on the other hand has a very high compute intensity. When one uses OpenMP to parallelize this edge-loop, a write-contention arises (during the scatter step) whenever the vertex (iPoint or jPoint) is shared by two edges that are assigned to different OMP threads. This write-contention is taken care of by either using OMP atomics or by pre-assigning the vertex to a particular "owner" thread.

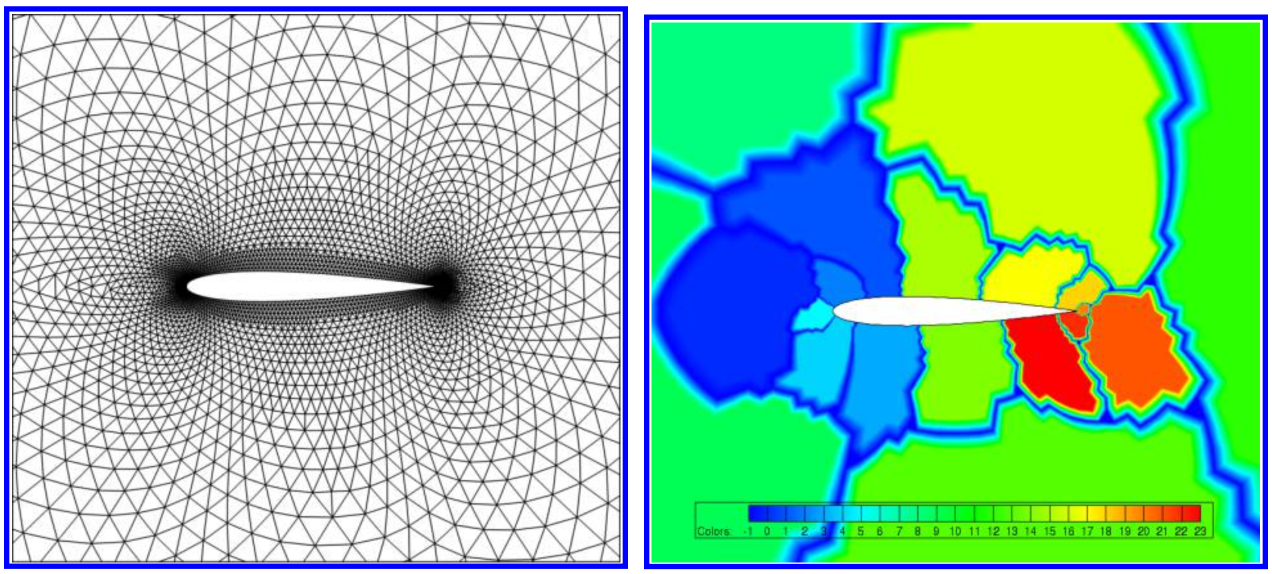
Listing 1. Typical Edge-loop in SU2

```
for (iEdge = 0; iEdge < geometry->GetnEdge(); iEdge++) {
{
  //Gather data from end-points of iEdge
   iPoint = geometry->edge[iEdge]->GetNode(0);
   jPoint = geometry->edge[iEdge]->GetNode(1);

   ProjVel_i = node[iPoint]->GetProjVel();
   ProjVel_j = node[jPoint]->GetProjVel();

   ...

   //Do some compute
   Mean_ProjVel = 0.5 * (ProjVel_i + ProjVel_j);
   Mean_SoundSpeed = 0.5 * (SoundSpeed_i + SoundSpeed_j) * Area;
   Lambda = fabs(Mean_ProjVel) + Mean_SoundSpeed;

   //Scatter the results to end-points of iEdge
   //Here we are updating Lambda at iPoint and jPoint
   //This is where there is write-contention as iPoint and/or jPoint are shared between different
       iEdges
   node[iPoint]->AddLambda(Lambda);
   node[jPoint]->AddLambda(Lambda);
}
```

In order to define the owner for each of the edges, we purse a decomposition approach that mimics the type of coarse-grain parallelism typically seen with distributed memory applications with MPI. This approach has been successfully implemented in the literature.[4,7] Rather than use a classic coloring technique, each thread is given a subdomain that results from a decomposition of the underlying edge graph. The METIS software package is used to complete the partitioning. An example for an unstructured NACA 0012 mesh is shown in Fig. 5.

Decomposing the edge graph balances work by evenly distributing edges while minimizing dependencies at shared nodes (the "edge cuts" of the edge graph). In order to eliminate contention at the shared nodes, all edges that touch a shared node are replicated on each thread that shares the node. The appropriate structures for these repeated edges are then added to the code to eliminate contention, and the result is similar to a halo layer approach in a distributed memory application. The subdomains can then be further reordered, vectorized, etc. The key concept is that there will be redundant compute (each thread computes quantities along all of the edges that it owns, including repeats), but only one of the threads will be considered the owner of repeated edges and perform data writes. A basic load balancing is performed by assigning ownership of repeated edges/nodes in a greedy manner to the thread with the least owned edges.

There arises one conflicting requirement. For optimal load-balancing, we would like to use one of the dynamic scheduling options in OpenMP (such as dynamic, guided, or auto). However, with dynamic scheduling it is not possible to pre-assign an "owner" thread and therefore one cannot get rid of atomics when doing the vertex update. Thus, the edge-loops for which load-balancing is critical we choose dynamic scheduling (in our case we choose auto OMP scheduling in particular), and for loops where there are large number of vertex updates (requiring too many atomics) we choose static scheduling using the pre-defined "owner" thread approach. In particular, for Set_UndLapl_DissSwitch we choose the latter approach and for the rest we do dynamic scheduling. This is because the Set_UndLapl_DissSwitch function requires atomic updates for all solution variables at the shared vertex, and hence, the total cost of atomics is very high. For Set-Max_EigenValue, we choose auto OMP scheduling because just two OMP atomics are required for every iEdge iteration. For Centered_Residual we again choose auto OMP scheduling as load-balancing is critically important for this function. To summarize, we use a combination of dynamic scheduling and static-scheduling for different edge-loops to optimize and balance the atomics cost and the load-imbalance cost. The performance gains obtained by adding dynamic scheduling are presented later in this section.


Colors: -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

(a) Zoom view of edges in an unstructured mesh around the NACA 0012 airfoil.

(b) Domain decomposition of the edge graph. Each color represents a single domain.

Figure 5. An example of the decomposition of the edge graph. Each OMP thread is assigned a subdomain.

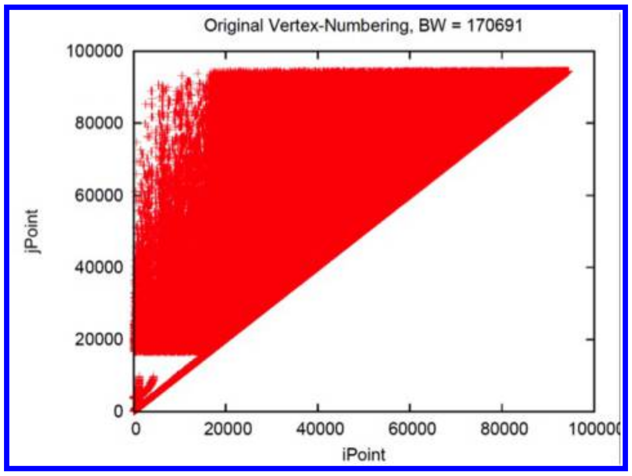
B. Improving Memory Performance

A number of approaches are typically followed when attempting to improve memory performance, and specific techniques will be described here as implemented in SU2_PHI. In general, the idea is to apply optimizations in order to improve the spatial and temporal locality of data. Three particular techniques for improving data locality are edge/vertex reordering, a smarter allocation for class objects, and lastly changing the data structures from array-of-structures (AOS) to structures-of-arrays (SOA) (see Ref.[36]). These are described next in detail.

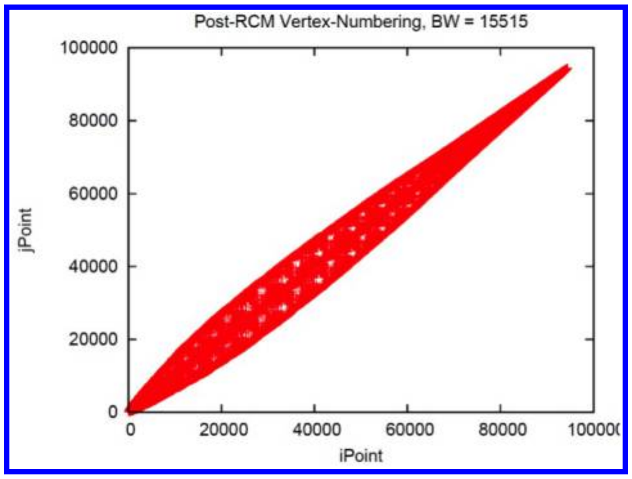
The first approach for memory optimization is a reordering of the nodes (unknowns) to minimize cache misses. This is accomplished via a Reverse Cuthill-McKee (RCM) algorithm that minimizes the bandwidth of the resulting problem. Edge renumbering has been performed based on a strategy in previous literature.[37] The mesh edges have been ordered according to points, i.e., the first edge-point (first index) increases monotonically as one progresses through the edges. We also minimize the jump in the second edge-point (second index). The idea is to keep the second edge-point that minimizes the jump with respect to the previous edge. In this manner, as the code traverses the edges and accesses the data at the edge end points, the memory is accessed in a cache-efficient manner, which leads to notable performance gains. In effect, by using RCM, we reduce the overall bandwidth of the adjacency matrix[38] of the unstructured mesh. An adjacency matrix essentially shows the edge connections in an unstructured mesh. The rows and columns of this matrix are vertices and a non-zero entry in the matrix means that the vertices are connected by an edge. This can be seen in Figs. 6(a) and (b) which show the adjacency matrix before and after the RCM transformation, respectively. The matrix bandwidth reduces from 170,691 to 15,515 by applying RCM re-numbering for the smaller tetrahedral ONERA M6 mesh.

The second approach for improving memory performance involves reworking some of the class structure in SU2_PHI in order to support more parallelization- and cache-friendly initializations of class data. For example, the CNumerics class has been modified so that the parent class is purely virtual with no class data, while the child classes allocate all of the data that is necessary for computing fluxes along the edges. This leads to a speed up of the code and also simplifies the parallelization of the flux loops using OpenMP. Another example is an improved memory allocation approach for the variables that are stored at each node (our unknowns) within the CVariable class. Here, we guarantee that memory for the objects is allocated in a contiguous array (in C-style), rather than using typical C++ allocations.

Lastly, another major memory optimization performed is a change in class structure from AOS to SOA. Listings 2 and 3 illustrate the basic concept. In the AOS case, the CSolver class contains a double pointer of an object of the CVariable class, node. The CVariable class contains the variables at a given vertex of the



(a) Before RCM



(b) After RCM

Figure 6. Effect of RCM re-numbering on the edge adjacency matrix of the ONERAM6 mesh.

Listing 2. Array of Structures

```
class CSolver {
   CVariable** node;
    ...
}
class CVariable {
   double* Solution;
   double* Undivided_Laplacian;
   double ProjVel;
   double Lambda;
    ...
}
```

mesh. In doing so, the variables (for example, Lambda) at neighboring vertices are not stored contiguously in memory. This increases the cache-miss rate as one traverses the edge-loop. On the other hand, in the SOA case, we do not make a reference to the object of the CVariable class. The CSolver class here contains pointers for each of the variables directly. These pointers store the corresponding variables for all the vertices of the mesh in a contiguous fashion. This reduces the memory access footprint and significantly improves the cache use efficiency. The typical edge-loop for an SOA case, looks like Listing 4 (in contrast with the AOS case in Listing 1). Note that, by implementing the AOS to SOA transformations, we do lose some flexibility in the code. One can no longer pick and choose a different type of the CVariable object to combine with a given CSolver class. Thus, there is a tension between performance and flexibility and one must balance these as required. In the present setup, the AOS to SOA transformations are coded using pre-processor directives such that the user can compile with AOS to SOA enabled if they desire better performance. The performance benefits of AOS to SOA are also quantified later in this section.

C. Vectorization

Vectorization is extremely critical for achieving high performance on modern CPUs as well as co-processors. The Intel® Xeon®, formerly codenamed "IvyBridge," processor is used in this study. In addition to scalar units, it has 4-wide double-precision (DP) SIMD units that support a wide range of SIMD instructions through Advanced Vector Extensions(AVX).[39] In a single cycle, they can issue a 4-wide DP floating-point multiply and add to two different pipelines. The Intel® Xeon Phi™ Co-processor, formerly codenamed "KnightsCorner," used in this study has 8-wide DP SIMD units for vector instructions. Thus, in theory, one

Listing 3. Structures of Arrays

```
class CSolver {
   double* Solution_all;
   double* Undivided_Laplacian_all;
   double* ProjVel_all;
   double* Lambda_all;
    ...
}
```

Listing 4. Typical Edge-loop in SU2 after AOS to SOA transformations

```
for (iEdge = 0; iEdge < geometry->GetnEdge(); iEdge++) {
{
   //Gather data from end-points of iEdge
    iPoint = geometry->edge[iEdge]->GetNode(0);
    jPoint = geometry->edge[iEdge]->GetNode(1);

    ProjVel_i = ProjVel_all[iPoint];
    ProjVel_j = ProjVel_all[jPoint];

    ...

    //Do some compute
    Mean_ProjVel = 0.5 * (ProjVel_i + ProjVel_j);
    Mean_SoundSpeed = 0.5 * (SoundSpeed_i + SoundSpeed_j) * Area;
    Lambda = fabs(Mean_ProjVel) + Mean_SoundSpeed;

    //Scatter the results to end-points of iEdge
    //Here we are updating Lambda at iPoint and jPoint
    //This is where there is write-contention as iPoint and/or jPoint are shared between different
        iEdges
    Lambda_all[iPoint] += Lambda;
    Lambda_all[jPoint] += Lambda;
}
```

can achieve a 4x and 8x speedup over scalar code for double-precision compute on a Xeon® processor and a Xeon Phi™ co-processor, respectively. Thus, vectorization is even more important for the co-processor.

Here, we describe the vectorization strategy for the Centered_Residual function that computes the convective flux residual using the JST scheme. As mentioned before, this function has a very high compute intensity and therefore is a great candidate for vectorization. We implement an outer-loop vectorization, where vectorization is achieved by computing on multiple edges simultaneously in multiple SIMD-lanes. The pseudo-code for the vectorized kernel is shown in Listing 5. Note that loop-tiling is done to implement vectorization. The first ivec loop computes the required input parameters for residual computation for a set of VECSIZE edges. The second ivec loop calls the ComputeResidual function on the set of VECSIZE edges simultaneously. The function ComputeResidual is a vector or "Elemental Function." Elemental Functions are a feature of the Intel® compilers,[40] and they are defined by adding a __attribute__((vector)) clause before the function definition.[40] The variable VECSIZE is made equal to the number of DP SIMD-lanes available in the architecture (4 for Xeon® and 8 for Xeon Phi™).

It is critically important that the parameters passed into the elemental function are accessed in a unit-strided way for different values of the loop iteration index (ivec). 1-D array (or single pointer) parameters, such as Neighbor_i/_j and Lambda_i/_j, are copied into static arrays of size VECSIZE as shown in the Listing 5. To achieve this for two-dimensional arrays (or double pointers), the inner-dimension of these arrays should be the ivec dimension, and its size must be equal to VECSIZE (for C language storage convention), as shown for Normal_vec. The linear clause in the elemental function definition implies that the variable varies linearly with the loop iteration index. The uniform clause specifies that these variables are constant for all the loop iterations and can be broadcasted once to all SIMD-lanes. For further description and additional clauses which are available in elemental functions, the reader is referred to the Intel Compiler Reference Manual.[40] The performance gains obtained by vectorizing this kernel are presented in the next sub-section.

D. Performance Results on Xeon® and Xeon Phi™

Now, we present the performance results obtained by adding all of the optimizations described above. The tests are performed on Xeon® and Xeon Phi™ (native execution). Native execution on Xeon Phi™ co-processor means that the code binaries are compiled for direct execution on the co-processor, and the host is not involved at all in the computation. The machine and tools configuration is given in Table 1.

Host with Xeon® Processor	Intel® Xeon® E5-2697v2 (formerly codenamed "IvyBridge") 2.70 GHz, 2 x 12 cores (dual-socket workstation), 64GB DDR3 1600 MHz RAM, Hyper-Threading (HT) enabled
Xeon Phi™ Co-processor	Intel® Xeon Phi™ C0-7120A, 1.238 GHz, 61 cores, 16 GB GDDR5 RAM, Turbo enabled
Tools	Intel® Composer XE 2015 (beta)

Table 1. Machine and tools configuration.

Figs. 7(a) and (b) show the speedup obtaind by adding various optimizations for Xeon® and Xeon Phi™, respectively. Results for both the small and large ONERA M6 meshes are shown. The simulation is run for 100 nonlinear iterations, and the time per iteration for the 100^{th} iteration is taken as the performance metric. Let's first look at the Xeon® results. The speedup is shown relative to the Base (MPI only) code. The Base code is run with 48 MPI ranks utilizing all 24 physical cores (48 ranks since HT is enabled). We first note that hybridization to MPI+OpenMP improves the performance by 1.11x for the small mesh. However, it does not make a difference for the large mesh. This is because the small mesh is more sensitive to memory latencies, and by hybridization, we are able to make better use of the shared-memory resources using OpenMP. For the large mesh there is enough compute to hide some of these latencies and hence there is not a significant speedup from hybridization. The hybrid runs are performed with 4 MPI ranks and 12 OMP threads per MPI rank (again utilizing all 24 physical cores). This same configuration is used for all the hybrid runs. As we add RCM, we see quite a big speedup for the small mesh. This is because RCM improves the cache-hit

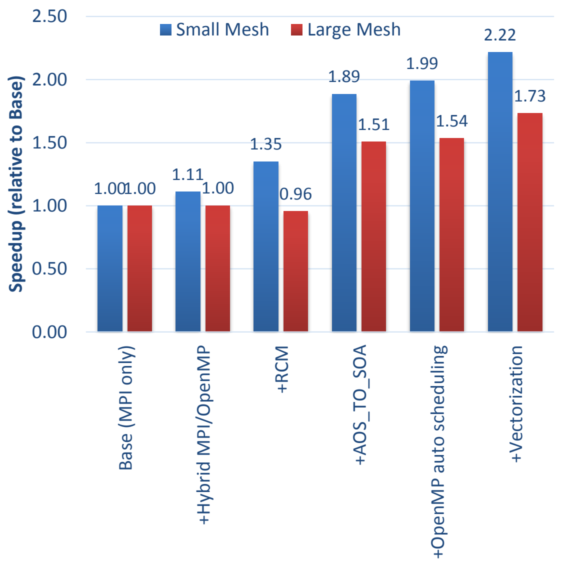
Listing 5. Pseudo-code for Centered_Residual and elemental function ComputeResidual to show vectorization

```
//Centered_Residual function
void Centered_Residual(CGeometry *geometry, ...) {

double Lambda_i_vec[VECSIZE];
double Lambda_j_vec[VECSIZE];
unsigned short Neighbor_i_vec[VECSIZE];
unsigned short Neighbor_j_vec[VECSIZE];
double Residual_vec[MAX_NVAR][VECSIZE];
double Normal_vec[3][VECSIZE]; //for 3-D and 2-D simulations
double *normal_temp;

for (iEdge = 0; iEdge < nEdges; iEdge += VECSIZE) {
   for (ivec = 0; ivec < VECSIZE; ++ivec) {
      //obtain variables required for computing the residuals
        normal_temp = geometry->edge[iEdge_vec]->GetNormal();
        for (iDim = 0; iDim < nDim; ++iDim) {
           Normal_vec[iDim][ivec] = normal_temp[iDim];
        }

       Neighbor_i_vec[ivec] = geometry->node[iPoint]->GetnNeighbor();
       Neighbor_j_vec[ivec] = geometry->node[jPoint]->GetnNeighbor();

       Lambda_i_vec[ivec] = Lambda_all[iPoint];
       Lambda_j_vec[ivec] = Lambda_all[jPoint];

       ...
     }

     //Call the elemental function on VECSIZE number of iEdges simultaneously
     #pragma simd vectorlength(VECSIZE)
     for (ivec = 0; ivec < VECSIZE; ++ivec) {
       numerics_local_vec[0]->ComputeResidual(ivec, Residual_vec, Normal_vec, Lambda_i_vec,
            Lambda_j_vec, Neighbor_i_vec, Neighbor_j_vec, ...);
     }
}

} //end of Centered_Residual

//ComputeResidual Elemental Function
__attribute__((vector(vectorlength(VECSIZE), linear(ivec:1), uniform(val_residual, val_normal,
    Lambda_i_vec, Lambda_j_vec, Neighbor_i_vec, Neighbor_j_vec, this))))
void ComputeResidual( int ivec, double val_residual[][VECSIZE], double val_normal[][VECSIZE],
    double Lambda_i_vec[], double Lambda_j_vec[], unsigned short Neighbor_i_vec[], unsigned short
    Neighbor_j_vec[], ...)
{
   \\Compute the Residual
}
```

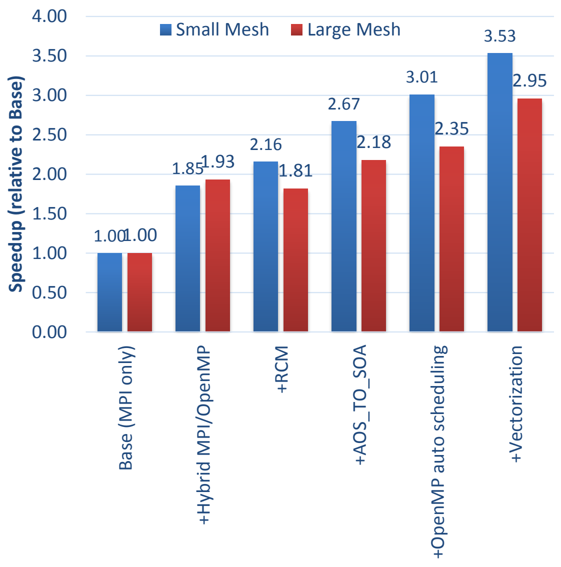
rate as described earlier. It is interesting to note that the speedup is actually negative for the large mesh case. This requires further investigation and is not within the scope of this paper. As we further add AOS to SOA transformations, a very noticeable jump in speedup is obtained for both small and large meshes. It can be seen that this is the most productive optimization. By adding auto OMP scheduling we get more speedup (more so for the small mesh again because it is more sensitive to load-imbalance among threads). Finally, by adding vectorization, we get about 10% overall speedup for both small and large meshes. Note that this is a significant gain from vectorization given the fact that only a single kernel (Centered_Residual) was vectorized. The other hotspots were not vectorized because of their low compute intensities in which case vectorization will not help much. After all the optimizations, 2.22x and 1.73x overall speedup was obtained over the Base (MPI only) code for small and large mesh, respectively.

Next, we present the Xeon Phi™ co-processor results. The code was executed natively on the co-processor with no host involvement. Overall, the picture for Xeon Phi™ looks similar to Xeon®. This is a big advantage of Xeon Phi™: one does not need to write and maintain different codebases for host and co-processor. The optimizations done to improve host performance help quite a bit in improving the co-processor performance and vice-verca.

In this case, the Base (MPI only) run was done using 240 MPI ranks utilizing 60 physical cores (240 ranks because Xeon Phi™ has a four hardware threads per core). The hardware threads help to hide latencies associated with the in-order instruction execution on the co-processor. Notice that there is a huge speedup obtained by hybridizing the code for both small mesh and large mesh. The speedup is much more than the corresponding numbers for Xeon®. For all hybrid runs, 4 MPI ranks and 60 OMP threads per MPI rank were used (utilizing 60 physical cores). Adding RCM speeds up the small mesh but slows down the large mesh case somewhat. This is similar to Xeon® results. Adding AOS to SOA transformations gives a good boost to speedup for both meshes. The AOS to SOA percentage speedup is less for Xeon Phi™ compared to Xeon®. Adding auto OMP scheduling helps a lot on Xeon Phi™ (more than Xeon®) due to the large number of OMP threads on Xeon Phi™. Further, addition of vectorization takes the total speedup for small mesh to 3.53x and large mesh to 2.95x, compared to the base (MPI only) case. Vectorization gains about 17% for small mesh and about 25.5% for the large mesh. This is more than double the gain compared to Xeon®, which is expected because of the twice-wide SIMD-lanes in Xeon Phi™compared to Xeon®. In summary, we note that hybridizing with good OpenMP scaling and vectorization are critically important to get performance out of the Xeon Phi™ co-processor.



(a) Intel® Xeon® results.



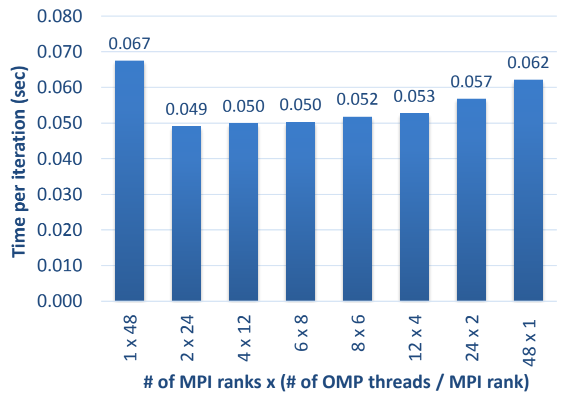
(b) Intel® Xeon Phi™ results.

Figure 7. Fine-grained single-node optimizations.

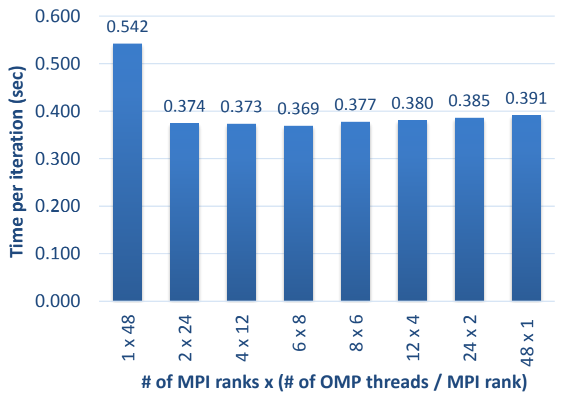
Finally, we look at absolute performance comparisons between Xeon® and Xeon Phi™. A full suite of hybrid runs are conducted on both architectures for small and large meshes. The number of MPI ranks is varied from 1 to 48 for Xeon® and 1 to 240 for Xeon Phi™, and the corresponding OMP threads per MPI rank is varied such that the product of no. of MPI ranks and no. of OMP threads per MPI rank is 48 for Xeon® and 240 for Xeon Phi™. This keeps the machines fully subscribed. Figs. 8 and 9 show the results for Xeon® and Xeon Phi™, respectively. These runtimes (time per iteration) are for the code which has all the optimizations presented earlier. A lower time per iteration signifies better performance. First, let's look at the Xeon® results in Fig. 8. The small mesh results show poor performance at either end of the spectrum. A sweet spot is obtained for 2 to 6 MPI ranks. The percentage of total variation in runtime (relative to largest runtime) for the small mesh is about 27% and for the large mesh is about 32%. For the large mesh, we note that the pure OpenMP mode (single MPI rank) performs very poorly compared to other configurations. There is also less variation in runtime if number of MPI ranks is more than one. Also, the best performance is obtained for 6 MPI ranks (with 8 OMP threads attached to each rank).

Next, we present the Xeon Phi™ results in Fig. 9. For both meshes, we see that the worst performance (highest runtime), by a large margin, is obtained for a pure MPI case (240 MPI ranks). For the small mesh, we see best performance for a pure OpenMP code (single MPI rank), whereas for large mesh, the best performance is obtained for 2 MPI ranks. The total amount of variation in runtimes (relative to largest runtime) is about 68% for the small mesh and 57% for the large mesh. This shows that Xeon Phi™ is much more sensitive to the hybrid run configuration compared to Xeon®. One needs to carefully choose the right configuration (which is different on the two architectures) for optimal performance. This is even more important when we run in symmetric mode (i.e. the workload is divided by placing MPI ranks on both Xeon® and Xeon Phi™simultaneously) to achieve a good load balance between them.

Note that for small mesh, the best Xeon® runtime is 0.049 sec, and the best Xeon Phi™ runtime is 0.157 sec (i.e., 3.2x of Xeon®). For the large mesh, the best Xeon® runtime is 0.369 sec, and the best Xeon Phi™ runtime is 0.991 sec (i.e., 2.69x of Xeon®). Thus, the absolute performance gap between Xeon® and Xeon Phi™ reduces as the mesh size increases. Even with a lower performance of Xeon Phi™ compared to Xeon®, the co-processor can still speedup the host run when added to the host by running in symmetric mode. For this, one must carefully divide the workload between host and the co-processor. Ongoing and future work is on further optimizing the Xeon Phi™ code.



(a) Small Mesh

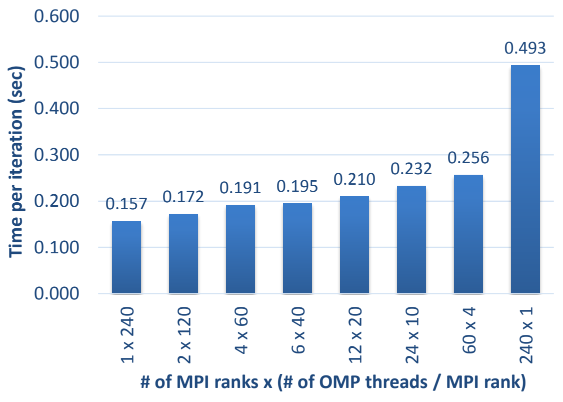


(b) Large Mesh

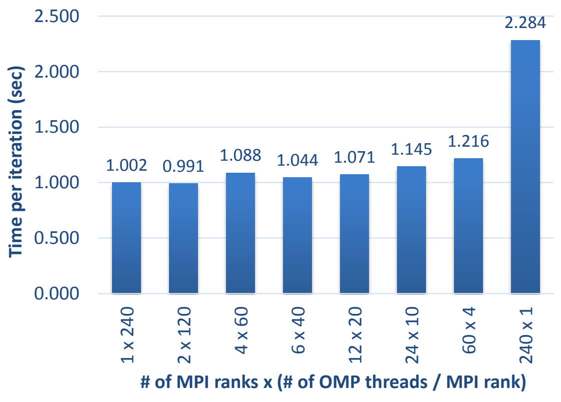
Figure 8. Intel® Xeon® Hybrid MPI/OpenMP runtimes for the small and large meshes.

V. Linear Solver Assessment for the Implicit Solution of the RANS Equations

In this section, we investigate a number of options and settings for linear solvers that are suitable for solving the RANS equations. More specifically, we seek a deeper understanding of the performance of a number of typical solvers in terms of robustness and scalability. The focus is placed on preconditioned Krylov-based methods (GMRES in particular), nonlinear multigrid methods with a variety of smoothers,



(a) Small Mesh



(b) Large Mesh

Figure 9. Intel® Xeon Phi™ Hybrid MPI/OpenMP runtimes for small and large meshes.

and a linear multigrid method. The goal is to discover the benefits and limitations of these solvers when calculating steady-state RANS solutions in serial and parallel.

A. Preconditioners for Krylov-based Solvers

The selection of a suitable preconditioner is critical to obtaining robust convergence with a Krylov-based linear solver. We have implemented and evaluated a variety of preconditioners for the GMRES algorithm. In particular, the objective has been to evaluate the performance of the different preconditioners in serial and parallel with local and global communication patterns that are found in realistic CFD problems.

Fig. 10 presents the results of an initial investigation into solver convergence with different preconditioners. Using a standard test case (transonic, turbulent flow around the RAE 2822 airfoil) and starting with serial calculations only, we can evaluate the number of internal iterations that are required to achieve a particular level of linear solver convergence for each preconditioner. Here, we apply the Jacobi, Linelet, LU-SGS, and ILU(0) preconditioners for a single nonlinear iteration (one time-step of the outer loop) of SU2 and count the number of iterations of the GMRES solver (inner loop) required to achieve a specified convergence tolerance.

The results follow traditional wisdom on the selection of a preconditioner: a trade-off exists between the amount of work required to apply the preconditioner and the subsequent convergence. On one end of the spectrum, the application of the simple Jacobi preconditioner is very cheap, but it requires the most linear solver iterations in all cases. Alternatively, the LU-SGS and ILU(0) preconditioners offer the best convergence properties at the cost of additional computational work. It is important to note that here we have not yet fully explored the impact of communication when these preconditioners are applied during distributed memory calculations with MPI. Each preconditioner requires at least one point-to-point communication (see Fig. 17) with nearest neighbors (due to the sweeping of the LU-SGS algorithm, it requires two point-to-point communications). These point-to-point communications can be removed so that the preconditioners operate in a purely local fashion, but this has a detrimental impact on stability and convergence. However, this is an area worth further investigation.

B. Nonlinear Multigrid Versus GMRES

During the course of this project, we have also focused on the multigrid (MG) technique as an alternative to a GMRES solver. With a proper implementation, it is expected that multigrid has better scaling potential than Krylov solvers, which are limited at large scale (1024+ compute nodes in a modern cluster) by the global reductions inherent within the orthogonolization step.[7,41,42] Multigrid involves only point-to-point communication to exchnage the boundary information across subdomains in a distributed memory setting, which scales very well on modern large scale clusters.

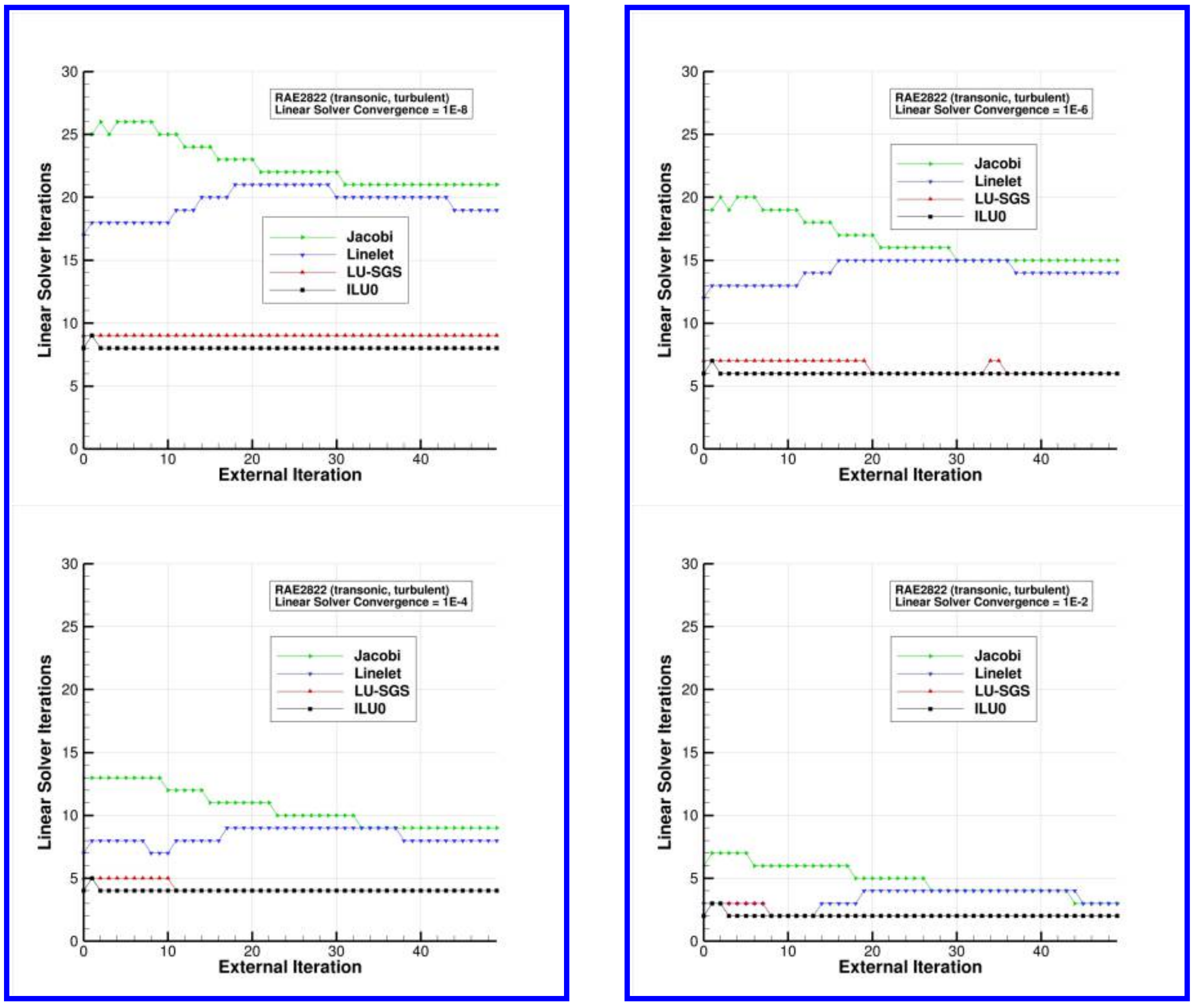


Figure 10. Performance of different preconditioners to achieve a particular convergence level (baseline case RAE 2822).

When the nonlinear multigrid algorithm drives the external iterations, the choice of smoother on each grid level plays an important role in the performance. The current implementation of SU2 can leverage both the Krylov-based methods as smoothers as well as versions of the preconditioners for the Krylov-based methods that have been repurposed as standalone smoothers for the linear system that arises on each grid level. In the following sections, only a single iteration of each smoother is applied. Studying the effects of multiple smoother iterations on each grid level is planned for future work. Additionally, a V-cycle is applied with the nonlinear MG, and the number of grid levels is defined automatically by ensuring that the rate of agglomeration between levels results in a sufficient reduction in the number of control volumes. Once this threshold is met, the agglomeration process is terminated.

In most cases, the differences in performance manifest themselves through restrictions on the maximum allowable CFL condition for maintaining numerical stability. The CFL condition governs the size of the time step for a given nonlinear iteration. As a direct consequence for obtaining steady solutions of the RANS equations, smoothers that require small CFL numbers will suffer from slow convergence.

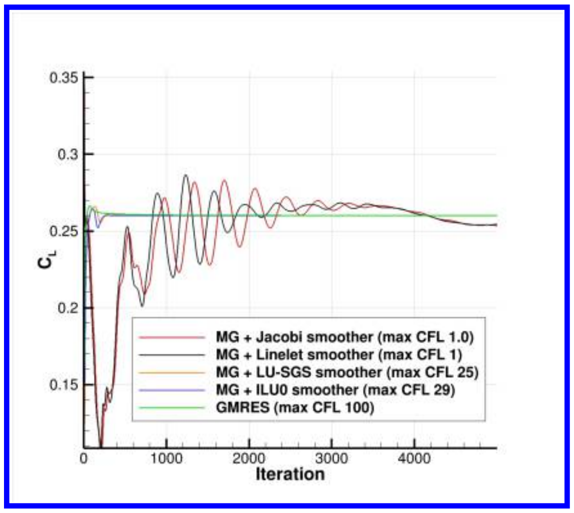


Figure 11. Comparison of GRMES convergence with non-linear multigrid using different smoothers.

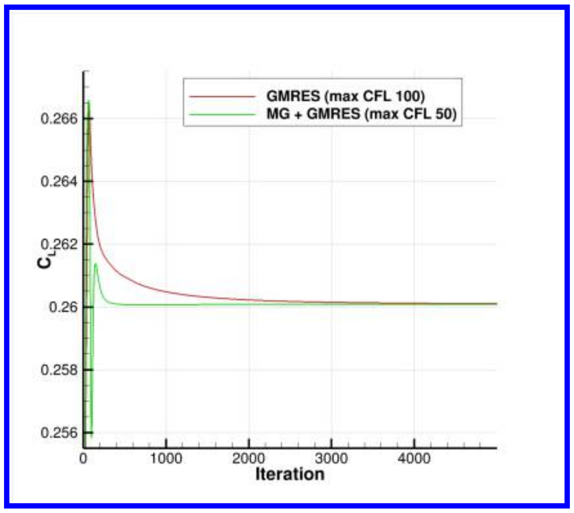


Figure 12. Effect of using a GMRES solver as the smoother in a non-linear multigrid iteration.

Fig. 11 contains a comparison of solver performance with MG for several smoother options against the performance of GMRES without using MG (single solution on the original fine grid). The performance metric in this case is to monitor the convergence of the lift coefficient to its steady value for the turbulent NACA 0012 simulation. First, it should be noted that in this serial setting, GMRES provides very robust convergence (the highest allowable CFL condition). It is also clear that the Jacobi and Linelet smoothers result in prohibitively slow convergence due to their CFL restrictions. However, nonlinear MG coupled with an LU-SGS or ILU(0) smoother may offer a potential alternative to GMRES, as these smoothers allow for higher CFL conditions that can compete with the convergence performance of GMRES.

GMRES can also be used as the smoother on each grid level for the nonlinear MG algorithm in SU2, and this was also exercised for the same NACA 0012 case, as shown in Fig. 12. Similar to the other smoothers, the CFL number must also be reduced for stability with a GMRES smoother when nonlinear MG is active. However, even with the reduced CFL, the overall convergence of the lift coefficient is improved, which demonstrates the potential of the MG algorithm. In general, we find that smaller CFL conditions are required for nonlinear MG, and one possible reason for this is the appearance of strong source terms within the FAS MG algorithm. A number of damping factors have been implemented as part of the nonlinear MG algorithm, and the application of these factors during restriction/prolongation can help alleviate stability issues at the cost of decreased convergence behavior.

Due to its potential as an alternative for GMRES, the ILU(0) smoother was exercised in single grid mode as well as with nonlinear MG in a similar numerical experiment. Fig. 13 shows a convergence comparison. In this case, nonlinear MG with ILU(0) enables a higher CFL condition as well as better overall convergence in the lift coefficient, which reinforces the idea that MG with an ILU(0) smoother holds potential.

Lastly, Fig. 14 presents a comparison of the lift convergence for the two top-performing smoothers with the nonlinear MG algorithm: GMRES and ILU(0). Here, we see that, while the maximum CFL number for

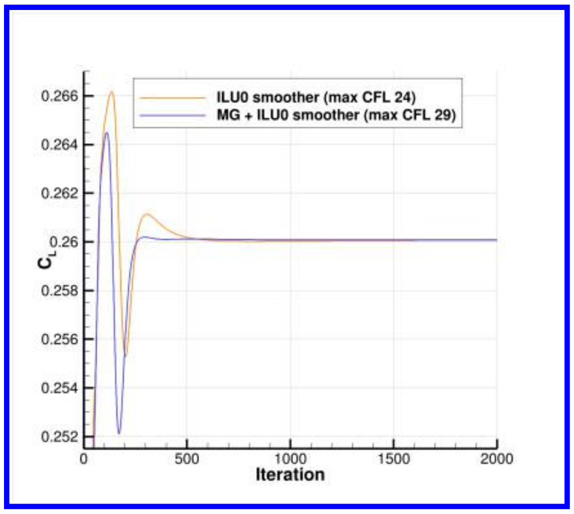


Figure 13. Non-linear multigrid compared with ILU0 smoother

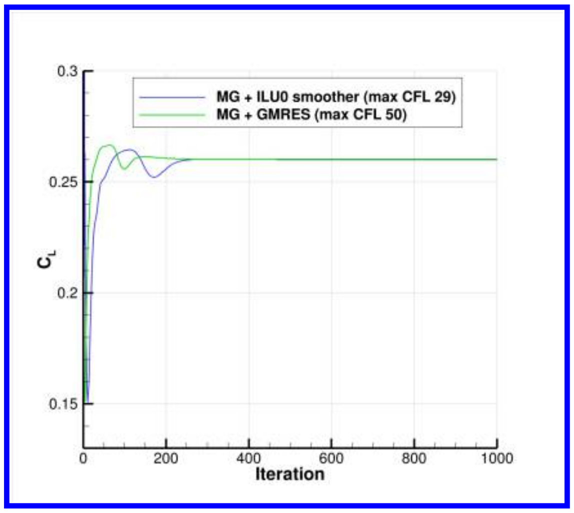


Figure 14. GRMES versus a ILU0 smoother including the non-linear multigrid in both simulations

the ILU(0) smoother is nearly half of that for GMRES, the overall convergence of the nonlinear solver is comparable.

C. Linear Multigrid

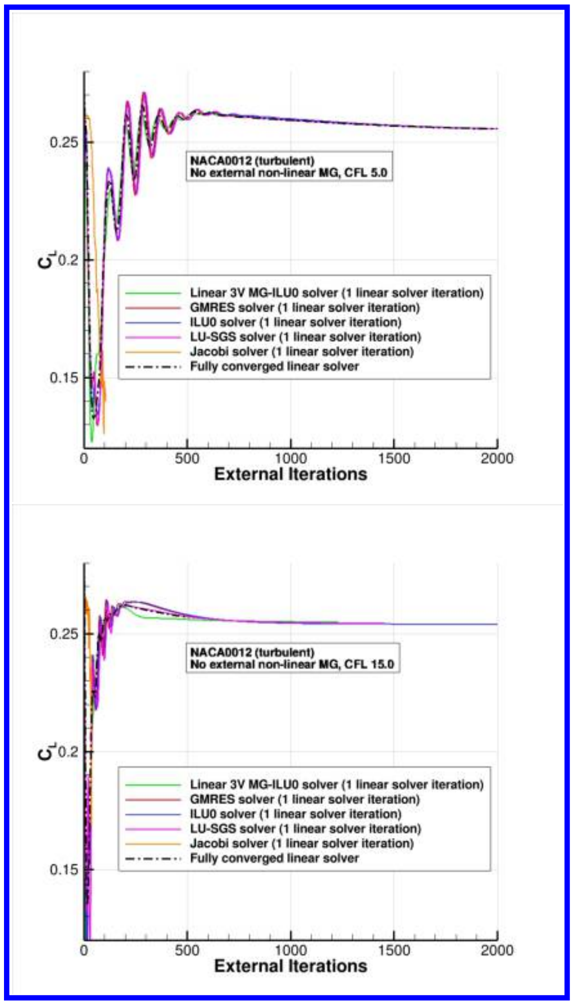
A linear multigrid algorithm has also been implemented in SU2 as an option to solve the linear system arising from the implicit solution of the RANS equations. The linear MG algorithm is also geometric in nature (the coarse grid levels are created using the same agglomeration algorithm as the nonlinear MG), but the key difference from the nonlinear MG is that the Jacobian matrix and residual vector (right-hand side) are computed once on each grid level prior to beginning the MG cycle and held fixed throughout the linear solve for each nonlinear time step (outer iteration). The same set of smoothers from the nonlinear MG are available for the linear MG algorithm.

The convergence performance of the linear multigrid method at several CFL conditions is compared to that of classical iterative solvers (smoothers) and a Krylov-based linear solver in Fig. 15. The behavior of all solver variants is compared against a baseline solution obtained by converging the linear system to machine tolerance with each nonlinear iteration. For moderate CFL numbers, the linear MG method with an ILU(0) smoother demonstrates comparable results to a GMRES solver with an ILU(0) preconditioner. While encouraging, these results were achieved on moderate grid sizes in serial. Therefore, the next step was to extend the linear MG implementation to parallel calculations on larger grids.

D. Robust Linear Multigrid

The initial implementation of linear multigrid suffered from stability issues on finer grids and when executed in parallel on multiple ranks (see Fig. 16). Two features were added in order to stabilize the linear MG solver: a damping factor that multiplies the restricted residual vector upon transferring grids and a time step limiter on the various grid levels (a limit is specified for the maximum allowable local time step in any control volume). With these additions, stability can be recovered with linear MG.

As shown in Fig. 18 for the NACA 0012 test case, the current implementation of the linear MG provides comparable convergence results to a GMRES solver for moderate CFL numbers. At higher CFL numbers, the Krylov-based methods remain more robust, which was a conclusion also drawn in our early investigations. However, we expect better scalability properties with the linear MG due to the lack of global communications. Future work is geared towards pushing the present algorithms to extreme scale, which will likely require further development of the MG algorithms in order to maintain robustness, including work in areas such as agglomeration strategies, damping, time step limits, and optimizing the amount of compute vs. communication across the grid levels.



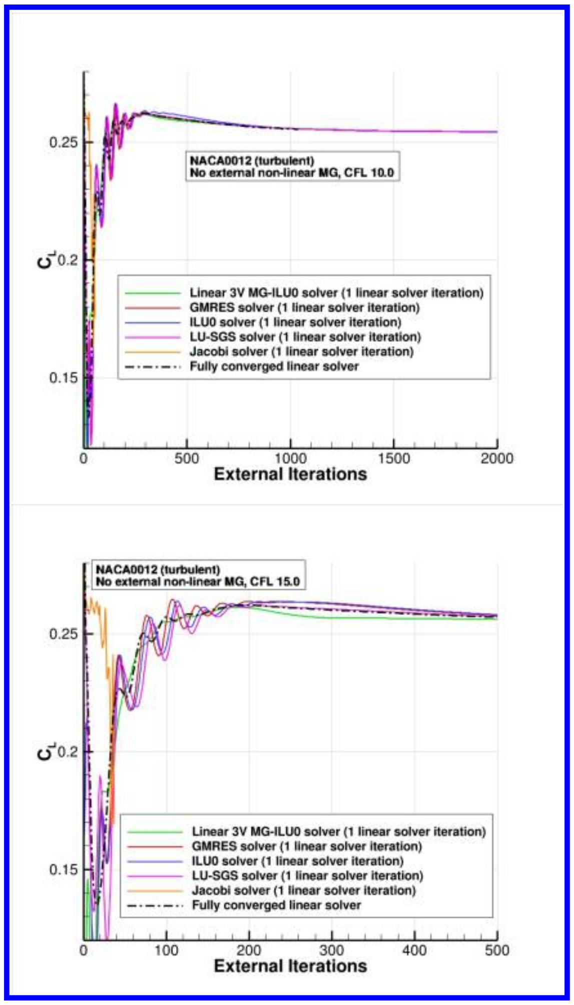


Figure 15. Linear multigrid compared with different alternatives at different CFL numbers.

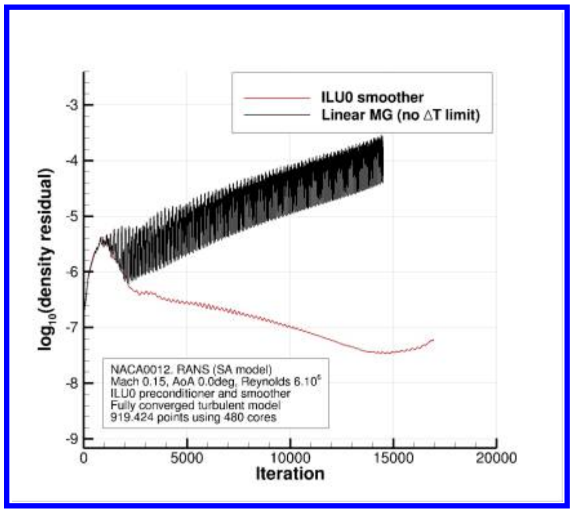


Figure 16. Divergence behavior of the original linear MG on fine grids in parallel

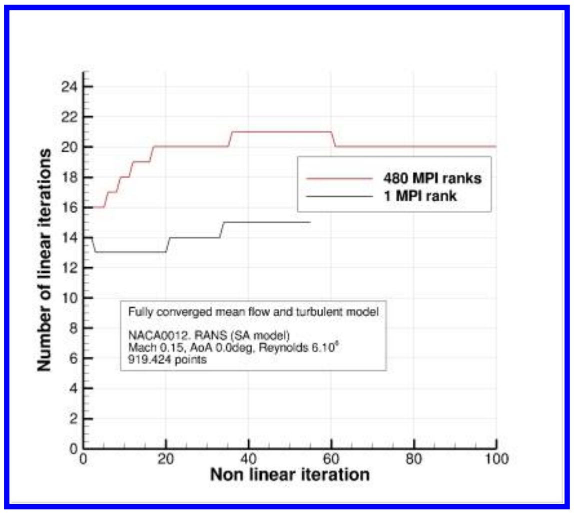


Figure 17. Parallel performance of the GMRES solver.

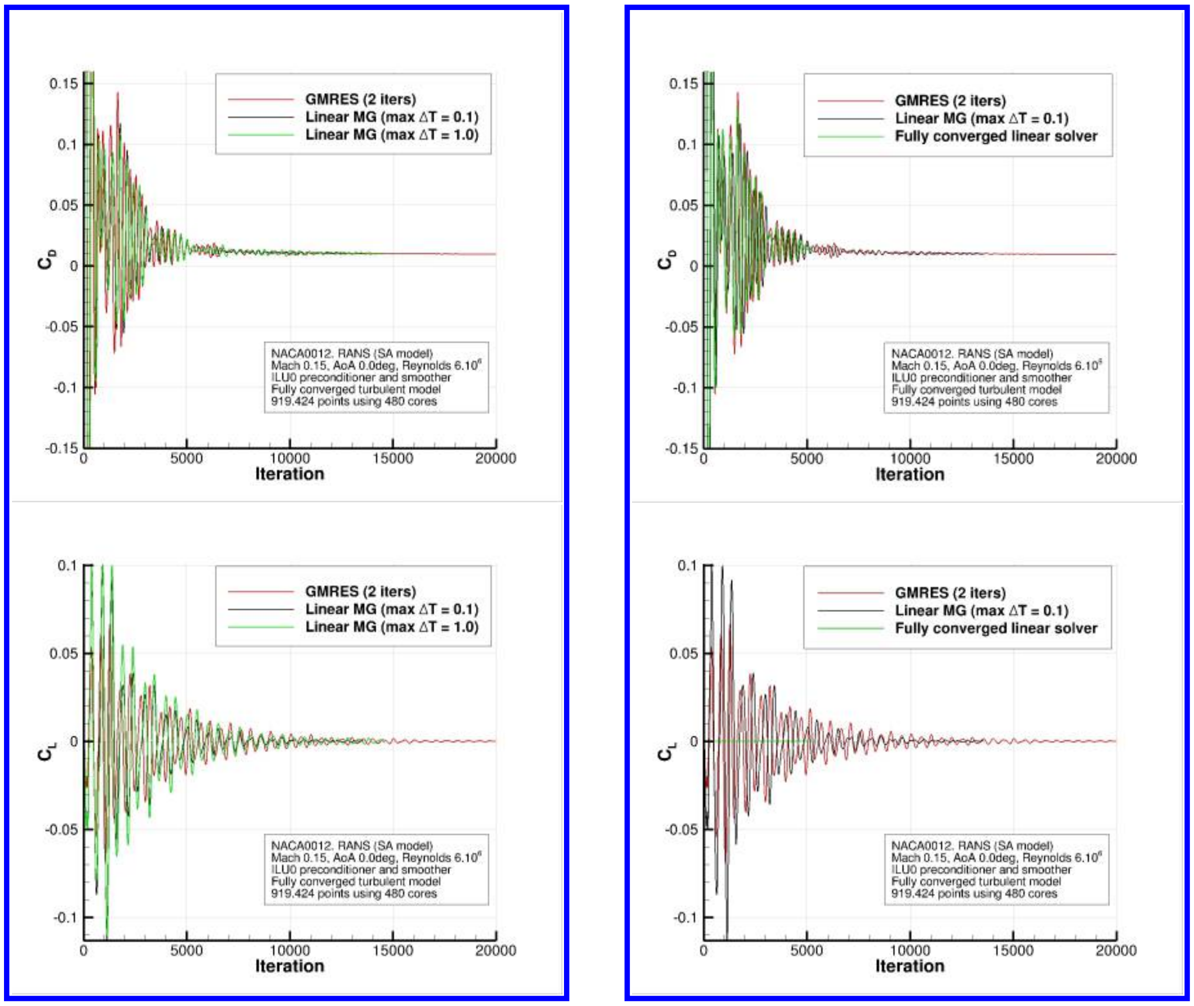


Figure 18. Drag and lift convergence using the new implementation of the linear multigrid algorithm.

VI. Conclusions

This article has presented the ongoing work towards optimizing the open-source SU2 analysis and design suite for execution on modern highly parallel (multi- and many-core) architectures. Particular emphasis has been placed on code parallelism (both fine- and coarse-grained), vectorization, efficient memory usage, and identifying the best-suited algorithms for modern hardware.

We have presented a variety of fine-grained optimizations that speed up the single node performance of SU2_PHI on Intel® Xeon® and Intel® Xeon Phi™ architectures. The optimizations are broadly classified into three categories: 1) Code hybridization into MPI+OpenMP, 2) Efficient memory use, and 3) Vectorization. Within these categories, multiple optimization strategies and their implementation details were presented. The performance gain from each of these optimizations was presented for both Intel® Xeon® and Intel® Xeon Phi™ architectures for a couple of meshes. We obtain a 2.22x and 1.73x speed-up over baseline code on Xeon® for the small and large meshes, respectively. And for Xeon Phi™, we obtain a speedup of 3.53x and 2.95x over baseline code for small and large meshes, respectively. One big advantage of Xeon Phi™ is that the optimizations done to improve Xeon® performance also help to improve Xeon Phi™ performance and vice-versa. This way, the developer does not have to maintain two different code bases for the host and the co-processor.

Absolute performance measurements were also presented for the Xeon® and Xeon Phi™ architectures. A full suite of hybrid runs were conducted by varying the number of MPI ranks across the entire spectrum possible. For the small mesh, the best Xeon Phi™ runtime is 3.2x of the best Xeon® runtime, and for the large mesh the best Xeon Phi™ runtime is 2.69x of the best Xeon® runtime. Thus the absolute performance gap between Xeon® and Xeon Phi™ is reduced as the mesh size increases. Even with the lower performance of Xeon Phi™ compared to Xeon®, the co-processor can still speedup the host run when added to the host by running in symmetric mode. For this, one must carefully divide the workload between host and the co-processor. Ongoing and future work is on further optimizing the Xeon Phi™ code.

The development of superior numerical algorithms and an optimal software implementation are both key ingredients to face the challenge of effectively leveraging future exascale machines. After the identification of some typical bottlenecks in the parallelization of a CFD code (e.g., linear solvers), we have proposed algorithms that only require point-to-point communications (a key aspect for promoting high parallel scalability). Furthermore, we have provided some arguments in support of the idea that choosing numerical methods that require smaller time steps for stability might pay off once the CFD simulation is pushed to extreme scale. In particular, we have studied the possibility of substituting Krylov-based methods (e.g., GMRES) by classical iterative schemes accelerated with a multigrid algorithm. Some very encouraging results have been presented in this area.

Finally, it should be reiterated that this article represents an early snapshot of promising results from ongoing efforts to optimize the SU2 platform in the above areas, and that the final goal of the proposed research is to combine the fine-grained optimizations of the codebase with scalable linear solver implementations in order to create a highly-optimized version of the SU2 suite for execution at extreme scale. Future work will address this goal, and all pertinent results and conclusions will be shared with the community.

VII. Acknowledgements

The first three authors would like to acknowledge the generous support from the Intel Corporation in the form of an Intel Parallel Computing Center at Stanford, the valuable time and energy of the engineers in the Intel Parallel Computing Laboratory, and additional support from the Intel Software & Services Group (SSG) Developer Relations Division (DRD), including access to Intel Corporation's Endeavour high-performance computing cluster in Rio Rancho, New Mexico. Gaurav Bansal would like to thank Paul Besl of Intel Corporation for many fruitful discussions.

References

[1]F. Palacios, M. R. Colonno, A. C. Aranake, A. Campos, S. R. Copeland, T. D. Economon, A. K. Lonkar, T. W. Lukaczyk, T. W. R. Taylor, and J. J. Alonso. Stanford University Unstructured (SU^2): An open-source integrated computational environment for multi-physics simulation and design. *AIAA Paper 2013-0287*, 51st AIAA Aerospace Sciences Meeting and Exhibit, January 2013.

[2]F. Palacios, T. D. Economon, A. C. Aranake, S. R. Copeland, A. K. Lonkar, T. W. Lukaczyk, D. E. Manosalvas, K. R.

Naik, A. S. Padron, B. Tracey, A. Variyar, and J. J. Alonso. Stanford University Unstructured (SU^2): Open-source analysis and design technology for turbulent flows. *AIAA Paper 2014-0243*, 2014.

[3]William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. High-performance parallel implicit {CFD}. *Parallel Computing*, 27(4):337 – 362, 2001. Parallel computing in aerospace.

[4]R. Aubry, G. Houzeaux, and M. Vasquez. Some useful strategies for unstructured edge-based solvers on shared memory machines. *AIAA Paper 2011-0614*, 2011.

[5]Dimitri J. Mavriplis and Karthik Mani. *Unstructured Mesh Solution Techniques using the NSU3D Solver*. American Institute of Aeronautics and Astronautics, 2014/12/22 2014.

[6]A.C. Duffy, D.P. Hammond, and E.J. Nielsen. Production level CFD code acceleration for hybrid many-core architectures. *Parallel Computing*, 2011.

[7]D. Mudigere, S. Sridharan, A.M. Deshpande, J. Park, A. Heinecke, M. Smelyanskiy, B. Kaul, P. Dubey, D. Kaushik, and D. Keyes. Exploring shared-memory optimizations for an unstructured mesh cfd application on modern parallel systems. In *Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2015.

[8]C. Hirsch. *Numerical Computation of Internal and External Flows*. Wiley, New York, 1984.

[9]D.C. Wilcox. *Turbulence Modeling for CFD*. 2nd Ed., DCW Industries, Inc., 1998.

[10]F.M. White. *Viscous Fluid Flow*. McGraw Hill Inc., New York, 1974.

[11]P Spalart and S. Allmaras. A one-equation turbulence model for aerodynamic flows. *AIAA Paper 1992-0439*, 1992.

[12]T. J. Barth. Aspect of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations. In *Lecture Notes Presented at the VKI Lecture Series*, 1994 - 05, Rhode Saint Genese Begium, 2 1995. Von karman Institute for fluid dynamics.

[13]A. Quarteroni and A. Valli. *Numerical approximation of partial differential equations*, volume 23 of *Springer series in computational mathematics*. Springer-Verlag Berlin Heidelberg New York, 1997.

[14]A. Jameson. A perspective on computational algorithms for aerodynamic analysis and design. *Progress in Aerospace Sciences*, 37:197–243, 2001.

[15]R. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Univesity Press, 2002.

[16]P. Wesseling. *Principles of Computational Fluid Dynamics*, volume 29 of *Springer Series in Computational Mathematics*. Springer-Verlag Berlin Heidelberg New York, 2000.

[17]A. Jameson. Analysis and design of numerical schemes for gas dynamics 1 artificial diffusion, upwind biasing, limiters and their effect on accuracy and multigrid convergence. *RIACS Technical Report 94.15, International Journal of Computational Fluid Dynamics*, 4:171–218, 1995.

[18]A. Jameson. Analysis and design of numerical schemes for gas dynamics 2 artificial diffusion and discrete shock structure. *RIACS Report No. 94.16, International Journal of Computational Fluid Dynamics*, 5:1–38, 1995.

[19]E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: a Practical Introduction*. Springer-Verlag, 1999.

[20]A. Jameson, W. Schmidt, and E. Turkel. Numerical solution of the Euler equations by finite volume methods using Runge-Kutta time stepping schemes. *AIAA Paper 1981-1259*, 1981.

[21]P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.

[22]M.-S. Liou and C. Steffen. A new flux splitting scheme. *J. Comput. Phys.*, 107:23–39, 1993.

[23]E. Turkel, V. N. Vatsa, and R. Radespiel. Preconditioning methods for low-speed flows. *AIAA*, 1996.

[24]T. J Barth and D. Jespersen. The design and application of upwind schemes on unstructured grids. *AIAA Paper 89-0366*, 1989.

[25]V. Venkatakrishnan. On the accuracy of limiters and convergence to steady state solutions. *AIAA Paper 1993-0880*, 1993.

[26]D. J. Mavriplis. Accurate multigrid solution of the Euler equations on unstructured and adaptive meshes. *AIAA Journal*, 28(2):213–221, February 1990.

[27]D. J. Mavriplis and A. Jameson. Multigrid solution of the Navier-Stokes equations on triangular meshes. *AIAA Journal*, 28(8):1415–1425, August 1990.

[28]P. Eliasson. Edge, a Navier-Stokes solver for unstructured grids. Technical Report FOI-R-0298-SE, FOI Scientific Report, 2002.

[29]N.A. Pierce and M.B. Giles. Preconditioned multigrid methods for compressible flow calculations on stretched meshes. *J. Comput. Phys.*, 136:425–445, 1997.

[30]O. Soto, R. Lohner, and F. Camelli. A linelet preconditioner for incompressible flow solvers. *Int. J. Numer. Meth. Heat Fluid Flow*, 13(1):133–147, 2003.

[31]D. J. Mavriplis. On convergence acceleration techniques for unstructured meshes. Technical report, Institute for Computer Applications on Science and Engineering (ICASE), 1998.

[32]Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[33]H. A. Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. and Stat. Comput.*, 13(2), 1992.

[34]D. J. Mavriplis. Multigrid techniques for unstructured meshes. Technical Report 95-27, ICASE, 1995.

[35]A. Borzi. Introduction to multigrid methods. Technical report, Institut für Mathematik und Wissenschaftliches Rechnen (Karl-Franzens-Universität Graz), 2003.

[36]P. J. Besl. A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel® Xeon® processors and Intel® Xeon Phi™ product family coprocessors. 2013.

[37]Rainald Lohner. Cache-efficient renumbering for vectorization. *Comm. Numer. Meth. Eng.*, 2008.

[38]I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Monographs on Numerical Analysis. Oxford University Press, 1989.

[39]Intel® architecture instruction set extensions programming reference. 2014.

[40]User and refernce guide for the Intel® C++ compiler 14.0. 2013.

[41]S. Williams, D.D Kalamkar, A. Singh, A.M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi-and manycore processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 96. IEEE Computer Society Press, 2012.

[42]J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D.D. Kalamkar, X. Liu, M.A. Patwary, Y. Lu, and P. Dubey. Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 945–955. IEEE Press, 2014.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2®, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.