Computers and Fluids 129 (2016) 146–158

Contents lists available at ScienceDirect

Computers and Fluids

journal homepage: www.elsevier.com/locate/compfluid

Performance optimizations for scalable implicit RANS calculations with SU2

Thomas D. Economon[a,*], Dheevatsa Mudigere[b], Gaurav Bansal[c], Alexander Heinecke[d], Francisco Palacios[e], Jongsoo Park[d], Mikhail Smelyanskiy[d], Juan J. Alonso[a], Pradeep Dubey[d]

[a] Department of Aeronautics & Astronautics, Stanford University, Stanford, CA, 94305, USA
[b] Parallel Computing Lab, Intel Corporation, Bangalore, India
[c] Software and Services Group, Intel Corporation, Hillsboro, OR, 97124, USA
[d] Parallel Computing Lab, Intel Corporation, Santa Clara, CA, 95044, USA
[e] Advanced Concepts Group, The Boeing Company, Long Beach, CA, 90808, USA

ARTICLE INFO

Article history:
Received 19 October 2015
Revised 26 January 2016
Accepted 7 February 2016
Available online 13 February 2016

Keywords:
CFD
Multigrid
Parallel algorithms
High performance computing

ABSTRACT

In this paper, we present single- and multi-node optimizations of SU2, a widely-used, open-source Computational Fluid Dynamics application, aimed at improving performance and scalability for implicit Reynolds-averaged Navier–Stokes calculations on unstructured grids. Typical industry-standard implementations are currently limited by unstructured accesses, variable degrees of parallelism, as well as the global synchronizations inherent in traditionally used Krylov linear solvers. Therefore, we rely on aggressive single-node optimizations, such as hierarchical parallelism, dynamic threading, compacted memory layout, and vectorization, along with a communication-friendly agglomeration (geometric) linear multigrid solver. Based on results with the well-known ONERA M6 geometry, our single core and shared memory optimizations result in a speedup of 2.6*X* on the latest 14-core Intel® Xeon™ [1] E5-2697v3 processor when compared to the baseline SU2 implementation with 14 MPI ranks. In multi-node settings, the hybrid OpenMP+MPI multigrid implementation achieves 2*X* higher parallel efficiency on 256 nodes over conventional Krylov-based (GMRES) methods.

1. Introduction

The solution of the compressible Reynolds-averaged Navier–Stokes (RANS) equations around complete aircraft configurations on computational meshes containing hundreds of millions of elements is now an everyday occurrence in the aerospace industry [1]. Efficient algorithms for Computational Fluid Dynamics (CFD), including implicit integration techniques and unstructured mesh technology, have allowed the aerospace and many other industries to perform large numbers of simulations for rapidly evaluating the performance of candidate designs, including those with increasingly complex geometries, such as the aircraft in Fig. 1.

CFD is both an important and a representative workload for high performance computing, frequently employed in benchmarking and the tuning of hardware and software programming environments. Furthermore, while most current and legacy CFD codes employ mesh partitioning strategies for distributed computing with MPI, only in recent years have practitioners started moving toward hybrid programming models, featuring both shared and distributed memory parallelism, in order to take advantage of new accelerator technologies, such as GPUs and co-processors. Even fewer codes explicitly address opportunities for fine-grained parallelism through vectorization (single instruction, multiple data, or SIMD). As modern hardware architectures move toward many-core designs with more available threads and additional vector lanes, extracting high performance requires effectively exploiting parallelism at multiple granularities simultaneously and making appropriate algorithmic choices.

An unstructured, implicit flow solver comprises a diverse range of kernels with varying compute and memory requirements, irregular data accesses, as well as variable and limited amounts of instruction-, vector- and thread-level parallelism, which make

[*] Corresponding author.
E-mail address: economon@stanford.edu (T.D. Economon).
[1] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

http://dx.doi.org/10.1016/j.compfluid.2016.02.003
0045-7930/© 2016 Elsevier Ltd. All rights reserved.

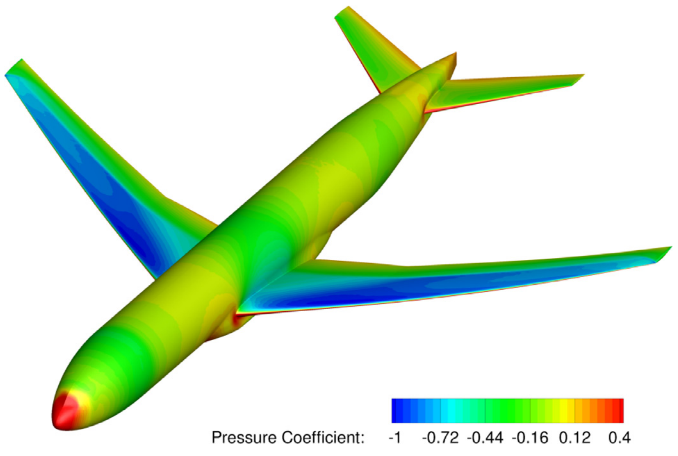

Fig. 1. Contours of pressure coefficient, as computed by SU2, on the surface of the NASA Common Research Model [2].

achieving high parallel efficiency a very challenging task. In particular, the linear solve required by implicit discretization schemes has consistently imposed a roadblock to scalability for many applications. In depth performance studies for such large-scale unstructured grid applications are still relatively limited and are a focus of active research [3–5]. 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.

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. Since 2012, we have developed and supported an open-source project, the SU2 software suite [8]: an open-source collection of software tools written in C++ for performing CFD analysis and design. It is built specifically for the analysis of partial differential equations (PDEs) and PDE-constrained optimization problems 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 aerodynamics, but through the initiative of users and developers around the world, SU2 is now being used for a wide variety of problems beyond aeronautics, including automotive, naval, and renewable energy applications, to name a few.

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 test-bed 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 to the community.

This paper makes the following contributions:

- We present the performance optimization of a complete unstructured, implicit CFD application.
- We explore a number of single-node optimization techniques and show significant performance benefits both at the kernel- and application-level. With our optimizations, we demonstrate a 2.6*X* speedup over a baseline running with a single rank per core on the current 14-core Intel Haswell processor.
- For distributed computing, we demonstrate that our optimizations continue to provide performance benefits as we scale and that, with a hybrid MPI+OpenMP implementation, we are able to push the strong scaling limit by 3.8*X*.
- A novel linear multigrid solver configuration improves scaling efficiency by 2*X* over conventional Krylov-based methods.
- The treatment of the open-source SU2 suite makes our work extensible to the larger CFD community for performing similar optimizations on modern, highly-parallel architectures.

This paper is organized as follows. We begin with necessary background information on the governing equations and discretization schemes in Section 2. In Section 3, we provide a detailed performance characterization. In Section 4, the experimental setup and the data sets used are described, as well as the performance profile of the baseline code. We present various key optimization strategies and performance results for the main computational kernels in Sections 5 and 6. Section 7 contains single-node and multi-node optimization results for the full SU2 application. We conclude in Section 8 and include a plan for future work.

2. Background

We begin with a brief overview of the governing equations and discretization schemes that will be necessary before discussing the specific implementation details and code optimizations later in the article.

2.1. Governing equations

We are concerned with compressible, turbulent fluid flows governed by the RANS equations, which can be expressed in differential form as

$$\begin{cases} \mathcal{R}(U) = \dfrac{\partial U}{\partial t} + \nabla\cdot\vec{F}^c - \nabla\cdot\vec{F}^v - Q = 0 & \text{in } \Omega \\ \vec{v} = \vec{0} & \text{on } S \\ \partial_n T = 0 & \text{on } S \\ (W)_+ = W_\infty & \text{on } \Gamma_\infty, \end{cases} \tag{1}$$

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

$$\vec{F}^c = \begin{Bmatrix} \rho\vec{v} \\ \rho\vec{v}\otimes\vec{v} + \bar{\bar{I}}p \\ \rho E\vec{v} + p\vec{v} \end{Bmatrix}, \quad \vec{F}^v = \begin{Bmatrix} \cdot \\ \bar{\bar{\tau}} \\ \bar{\bar{\tau}}\cdot\vec{v} + \mu^*_{tot} c_p \nabla T \end{Bmatrix}, \tag{2}$$

and $Q = \{q_\rho, \vec{q}_{\rho\vec{v}}, q_{\rho E}\}^\mathsf{T}$, 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

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

In 3D, the first line of Eq. (1) is a set of five coupled, nonlinear PDEs that are statements of mass (1), momentum (3), and energy (1) conservation in a fluid. The remaining lines of Eq. (1) represent the boundary conditions on no-slip aerodynamic surfaces S and far-field boundaries Γ_∞ that mimic the fluid behavior at infinity [9].

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 [10], which states that the effect of turbulence can be represented as an increased viscosity, the total viscosity is divided into laminar and turbulent components, or μ_{dyn} and μ_{tur}, respectively. In order to close the system of equations, the dynamic viscosity μ_{dyn} is assumed to satisfy Sutherland's law [11] (a function of temperature

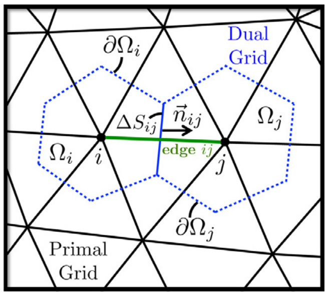

Fig. 2. Dual mesh control volumes surrounding two vertices, i and j, in the domain interior.

alone), the turbulent viscosity μ_{tur} is computed via a turbulence model, and

$$\mu_{tot} = \mu_{dyn} + \mu_{tur}, \quad \mu^*_{tot} = \frac{\mu_{dyn}}{Pr_d} + \frac{\mu_{tur}}{Pr_t}, \tag{4}$$

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 mean flow state U and a set of new variables for the turbulence. The Spalart–Allmaras (S–A) model [12] will be used for the numerical experiments in this article, which requires the solution of an additional scalar PDE.

2.2. Numerical solution of the RANS equations

As a necessary background for discussing the performance optimizations, the following sections contain a short description of the numerical discretization schemes for solving the RANS equations in SU2.

2.2.1. Spatial integration via the finite volume method

In SU2, we discretize in space using a finite volume method (FVM) [13–20] with a standard edge-based data 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 primal cells sharing the particular vertex. A notional example of a triangular primal grid and its dual counterpart under this construction is shown in Fig. 2.

After integrating the governing equations over a control volume and applying the divergence theorem, one obtains the semi-discretized, integral form:

$$\begin{aligned} 0 &= \int_{\Omega_i} \frac{\partial U}{\partial t}\, d\Omega + \sum_{j\in\mathcal{N}(i)} (\tilde{F}^c_{ij} + \tilde{F}^v_{ij})\Delta S_{ij} - Q|\Omega_i| \\ &= \int_{\Omega_i} \frac{\partial U}{\partial t}\, d\Omega + R_i(U), \end{aligned} \tag{5}$$

where $R_i(U)$ is the numerical residual that represents the integration of all spatial terms for the control volume surrounding vertex i. $\tilde{F}^c_{ij}$ and $\tilde{F}^v_{ij}$ are the numerical approximations of the convective and viscous fluxes projected along an edge, respectively, and Q is the 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 vertices to vertex i. These quantities are also labeled in Fig. 2.

The convective and viscous fluxes are evaluated at the midpoint of an edge. The convective fluxes can be discretized using centered or upwind schemes in SU2. In this article, we focus exclusively on the approximate Riemann solver of Roe [21] for computing the convective terms with second-order reconstruction via the MUSCL approach [22]. Slope limiting is applied to preserve monotonicity in the solution by limiting the gradients during higher-order reconstruction, and the Venkatakrishnan [23] limiter is chosen for this. The convective term for the scalar variable in the S–A turbulence model is discretized using a first-order upwind scheme.

In order to evaluate the viscous fluxes with a finite volume method for both the mean flow and turbulence model, flow quantities and their first derivatives are required at the faces of the control volumes. The spatial gradients of the flow variables are calculated in a pre-processing step at all vertices using a Green–Gauss approach [24] and then averaged to obtain these gradients at the cell faces when computing viscous fluxes along the edges. Source terms are approximated at each vertex using piece-wise constant reconstruction within each of the dual control volumes.

In practice, the numerical residual $R_i(U)$ at each vertex from Eq. (5) is evaluated with each nonlinear iteration using a sequence of loops over the edges and vertices:

1. Pre-processing loops over edges to compute quantities needed for the flux routines, such as gradients and limiters.
2. A loop over all of the edges in the primal mesh in order to calculate the convective and viscous fluxes along with their Jacobians for implicit calculations (to be discussed next).
3. A loop over all of the vertices in the primal mesh and compute source terms in each dual control volume given the current state.
4. A loop over all of the boundary vertices in the primal mesh in order to impose boundary conditions.

This series of steps results in a value of $R_i(U)$ at each vertex at a particular instance in time, which can then be substituted into Eq. (5) and integrated in time to arrive at either a steady state or a time-accurate solution for the state vector U. These edge loops will be a major focus of the performance optimizations to be discussed.

2.2.2. Time integration

We now consider the techniques for time-marching the coupled system of ordinary differential equations for the flow problem represented in Eq. (5), which can be rewritten as

$$\frac{d}{dt}(|\Omega_i|U_i) + R_i(U) = 0, \tag{6}$$

where $|\Omega_i| = \int_{\Omega_i(t)} d\Omega$. By discretizing the time derivative term, one obtains a fully-discrete finite volume form of the governing equations.

For particularly stiff problems, implicit methods can be used to improve convergence due to their increased numerical stability. Here, we use the backward Euler scheme, where the residual is evaluated using the solution state at the new time level U^{n+1}. Applying this to Eq. (6), one has

$$|\Omega_i|\frac{\Delta U_i}{\Delta t_i} = -R_i(U^{n+1}), \tag{7}$$

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. Therefore, a first-order linearization about time level n is performed:

$$R_i(U^{n+1}) = R_i(U^n) + \sum_{j\in\mathcal{N}(i)} \frac{\partial R_i(U^n)}{\partial U_j}\Delta U_j^n + \mathcal{O}(\Delta t^2). \tag{8}$$

Introducing Eq. (8) into Eq. (7), we find that the following linear system should be solved to find the solution update (ΔU_i^n):

$$\left(\frac{|\Omega_i|}{\Delta t_i^n}\delta_{ij} + \frac{\partial R_i(U^n)}{\partial U_j}\right)\cdot \Delta U_j^n = -R_i(U^n), \tag{9}$$

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

$$\frac{\partial R}{\partial U} := \frac{\partial R}{\partial U} + \begin{bmatrix} \ddots & & & & \\ & \frac{\partial \tilde{F}_{ij}}{\partial U_i} & \cdots & \frac{\partial \tilde{F}_{ij}}{\partial U_j} & \\ & \vdots & \ddots & \vdots & \\ & -\frac{\partial \tilde{F}_{ij}}{\partial U_i} & \cdots & -\frac{\partial \tilde{F}_{ij}}{\partial U_j} & \\ & & & & \ddots \end{bmatrix}. \tag{10}$$

Implicit methods enable the use of higher CFL conditions than with explicit methods, which translate to the specific values of Δt_i 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. Allowable local time-step values can be calculated from an estimation of the convective and viscous spectral radii at every vertex in the mesh [25].

2.3. Agglomeration linear multigrid solver

With each nonlinear iteration of the RANS solver, the system in Eq. (9) is solved to some tolerance to provide a solution update. Common linear solver choices for modern CFD solvers include classic iterative methods and preconditioned Krylov subspace methods. While a number of linear solvers and preconditioners are available in SU2, previous work [26] established that two solver configurations exhibited good convergence behavior and were ideal for further performance investigation: the Generalized Minimal Residual (GMRES) method [27] with an Incomplete LU (with no fill in, i.e., ILU(0)) preconditioner, and a geometric linear multigrid method (LMG) with a single iteration of an ILU(0) smoother on each level.

In this particular in-house implementation, a geometric agglomeration multigrid method has been used that is suitable for unstructured meshes. This strategy consists of choosing a seed point (a vertex in a vertex-based code such as SU2) that 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 manually create independent meshes for the coarse levels, as this task is completely automated in SU2.

While multigrid methods are commonly applied in a non-linear fashion for acceleration the convergence of the flow equations (i.e., FAS multigrid), we will be applying the geometric multigrid algorithm as a linear solver for the system generated with each non-linear (outer) iteration. In practice with LMG for this article, the residual vector and Jacobian matrix are computed just once on each mesh level at the beginning of a nonlinear iteration of the equations and are held fixed during the smoothing and traversal of the multigrid levels in a V pattern (the typical V-cycle) to provide an update to the flow state.

3. Computational patterns and optimization challenges

The SU2 kernels can be broadly classified into the following four categories, which have very distinct computational patterns:

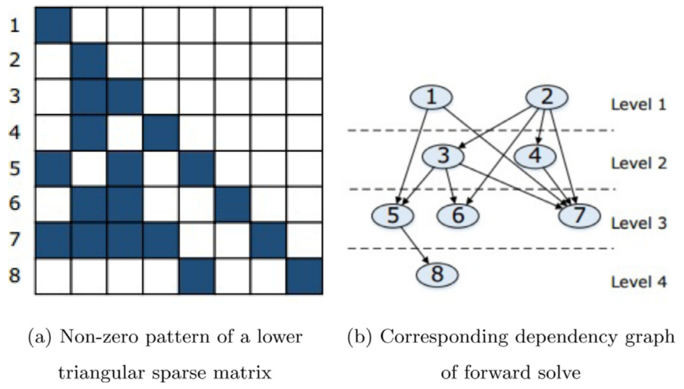
1
2
3
4
5
6
7
8
(a) Non-zero pattern of a lower triangular sparse matrix
Level 1
Level 2
Level 3
Level 4
(b) Corresponding dependency graph of forward solve

Fig. 3. Sparse recurrences.

1. Sparse, narrow-band recurrences: approximate factorization and back-substitution.
2. Edge-based “stencil op” loops: flux computations, gradients, limiters, and Jacobian term evaluations.
3. Vertex-based loops: state vector and auxiliary vector updates.
4. Global collectives: vector inner products and norms.

This classification holds for unstructured CFD applications in general [7,28,29], and most of these workloads consist of the same computational patterns.

A majority of the execution time is expected to be spent in the recurrences (mostly linear algebra) and the edge-based loops containing the physics of the application. Our profiling of the baseline code, described later, shows that these operations together account for 96% of the overall execution time. The collectives typically have very little floating point work and involve a logarithmically deep succession of messages to traverse the sub-domains 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, becoming the primary scaling bottleneck. However, the shared-memory challenges are different, and are described in the next section for both edge-based loops and recurrences.

3.1. Sparse, narrow-band recurrences

The recurrences have limited parallelism, proportional to the number of independent edges. Furthermore, the flop/byte ratio for these recurrences is usually low, and these operations are expected to be bandwidth-bound. The primary challenges with these operations are:

1. Extracting sufficient parallelism. This is a key challenge since the available parallelism is limited by the number of levels (or wave-fronts) in the task dependency graph, as show in Fig. 3b.
2. Load imbalance and synchronization overhead. An irregular sparsity pattern (see Fig. 3a) can result in load imbalance, while limited parallelism can expose the overhead of inter-core synchronization.

3.2. Edge-based loops

These loops predominantly occur for the evaluation of fluxes, gradients, limiters, Jacobian terms, and Jacobian-vector products. Typically, these loops have color-wise concurrency and nearest neighbor communication to complete the edges cut by the mesh partitioning. These loops often involve significant computational work per pair of vertices that share an edge, and they comprise the

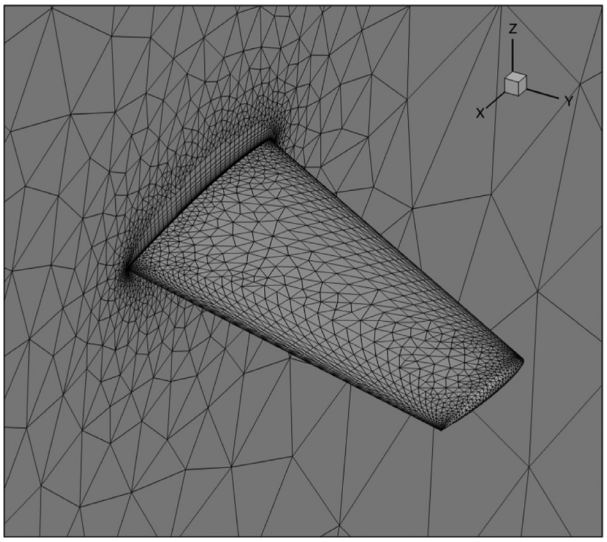

Fig. 4. View of the coarse ONERA M6 mesh near the wing and symmetry plane.

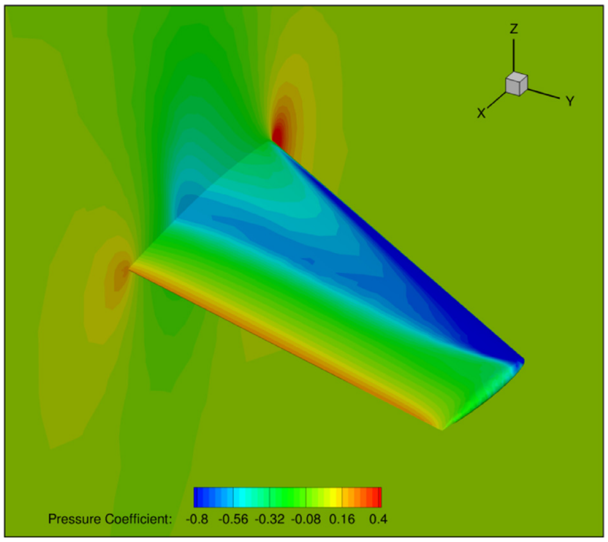

Fig. 5. Pressure coefficient contours for the coarse ONERA M6 mesh.

bulk of the floating-point operations. The key optimization challenges for these kernels are:

1. Extracting thread- and SIMD-level parallelism in the presence of loop-carried dependencies due to vertices shared by multiple edges.
2. Exploiting SIMD-level parallelism in the presence of irregular memory accesses.

4. Numerical experiment setup

In this section, the inputs for our numerical experiments are described, including chosen test cases, hardware, and a baseline code profile definition.

4.1. Data sets

For our numerical experiments, we have chosen to compute turbulent flow around the ONERA M6 wing geometry. This classic test case was designed in 1972 by the ONERA Aerodynamics Department as an experimental geometry for studying three-dimensional, high Reynolds number flows with some complex flow phenomena (transonic shocks, shock-boundary layer interaction, and separated flow, for instance).

The chosen flow conditions are from Test 2308 from the experimental data by Schmitt and Charpin [30]: a Mach number of 0.8395, an angle of attack of 3.06 degrees, and an angle of side-slip of 0.0 degrees. These correspond to a Reynolds number of 11.72 million based on the mean aerodynamic chord of 0.64607 m. These transonic, high Reynolds number conditions are representative of an aircraft in cruise and will exercise all of the flow physics and numerical algorithms of interest.

Coarse and fine meshes were used for the numerical experiments. The coarse mesh was composed of 126,539 vertices and 728,076 tetrahedra, while the fine mesh was composed of 9,252,922 vertices and 54,821,473 tetrahedra. The mesh spacing near the wing surface was set to achieve the proper resolution of turbulent behavior ($y^+ \approx 1$). A no-slip, adiabatic condition is satisfied on the wing surface, a symmetry plane is used to reflect the flow about the plane of the root wing section to mimic the effect of the full wing planform, and a characteristic-based condition is applied at a spherical far-field boundary. The surface mesh for the wing geometry and symmetry plane are shown in Fig. 4, and Fig. 5 presents pressure contours on the surface of the wing on the same coarse mesh.

4.2. Platforms used for experiments

For the single-node experiments, we used a workstation with two Intel Xeon E5-2697 v3 processors (each processor has 14 cores), running at a clock speed of 2.6 GHz. The architecture features a super-scalar out-of-order micro-architecture supporting 2-way hyper-threading, which results in a total of 28 hardware threads per socket. In addition to scalar units, it has 4-wide, double-precision SIMD units that support a wide range of SIMD instructions through Advanced Vector Extensions v2 (AVX2) [31]. In a single cycle, two 4-wide double-precision floating-point "multiply and add" instructions can be issued, as two different execution units are available.

Each core is backed by a 32KB L1 and a 256KB L2 cache, and all cores share a 35MB last level L3 cache. Together, the 14 cores can deliver a peak performance of 515.2 Gflop/s of double-precision arithmetic using AVX2. The workstation has 64GB DDR4 memory. Each socket has four channels running at 1867 MHz that can deliver a STREAM bandwidth of 50 GB/s.

For the multi-node experiments, we used Intel's Endeavor supercomputer. Each node on the cluster consists of two sockets, and each socket houses the same Intel Xeon E5-2697 v3 processor as detailed for the single-node experiments. The nodes are connected via the Infiniband FDR-14 interconnect. The Intel Parallel Studio XE 2015[2] toolchain was used for compilation and its MPI library.

4.3. Baseline code performance

A stripped down a version of the SU2 suite, with additional profiling hooks to allow for more accurate performance measurements, serves as our baseline code. For single-node studies, we compare performance against the baseline on both a single core and also with flat-MPI (i.e., MPI alone with a single rank per core). The latter is necessary due to the lack of multi-threading support in the original suite, which is one of the key contributions of this work. The baseline version running on 14 MPI ranks utilizes all available compute resource to ensure a fair comparison. However, it is important to note that, from an algorithmic point of

[2] https://software.intel.com/en-us/intel-parallel-studio-xe

view, increasing the number of MPI ranks is not necessarily desirable, as the resulting partitioning affects the convergence behavior and scaling potential of the code, and this will be discussed in later sections.

Fig. 7 shows the performance profile of the baseline code using the coarse ONERA M6 mesh. The kernels can be classified under three main categories:

1. The linear solver. This includes both the agglomeration multigrid solver used for the Navier–Stokes equations (mean flow) and the preconditioned GMRES solver used for the turbulence equation. Together, these contribute the largest chunk of the execution time at 52%.
2. The flux and Jacobian calculations (spatial integration) are second with a contribution of 25%.
3. Finally, the gradient, limiters, and other auxiliary computations needed to complete the spatial and temporal integration schemes make up 17% of the execution.

Each category here consists of 2 or 3 key kernels, and this handful of kernels contribute to approximately 94% of the total execution time. Hence, these kernels are the primary focus of the current work.

5. Fine-grained parallelism and shared memory optimizations

In this section, we describe various shared memory and multi-threading optimization strategies as applied to each of the compute patterns described above. We focus on sparse recurrences (linear solver) and the edge loops, since these are the computational patterns contributing to the majority of the execution time. The discussion is divided into the different classes of optimizations, but first, we will detail our general approach to multi-threading.

5.1. General multi-threading approach

We have implemented a high-level, functional OpenMP (OMP) approach for employing multiple threads during execution. This approach involves a single OMP parallel region at a very high level in the program, i.e., a single fork-join around the main external iteration loop of the solver. Here, the iteration space for edge loops and point updates is divided such that each OMP thread is assigned ownership of a sub-domain of edges and vertices, which mimics the style of coarse-grained parallelism typically seen with distributed memory CFD applications through MPI. This approach is different than a classic edge coloring scheme, and it has been successfully implemented in the literature [4,7].

In this approach, the sub-domains result from a decomposition of the underlying edge graph of the unstructured mesh. The METIS software package is used to complete the partitioning [32]. Decomposing the edge graph balances work by evenly distributing edges while minimizing dependencies at shared vertices, which become the “edge cuts” of the edge graph. In order to eliminate contention at the shared vertices, all edges that touch a shared vertex are replicated on all threads that share the vertex. The appropriate data 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 sub-domains can then be further reordered, vectorized, etc.

The key concept in this approach is that each thread computes quantities along all of the edges that have been assigned to it (including repeats), but only one of the threads will be considered the owner of each repeated edge. Only the owner thread performs data writes to the adjacent vertices of a repeated edge. In this manner, no additional communication or reduction of data across threads is necessary, and OMP atomics can be avoided entirely at the cost of a small overhead due to redundant compute along repeated edges.

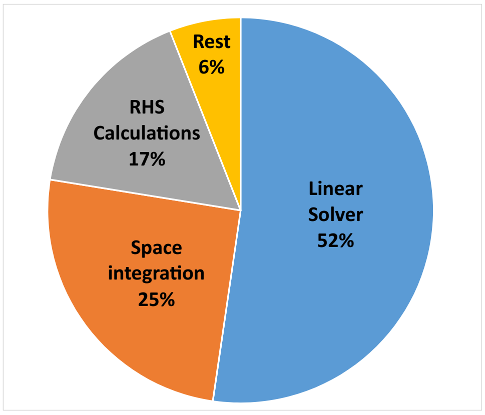

Fig. 6. An example of the decomposition of the edge graph. Each OMP thread is assigned a sub-domain.

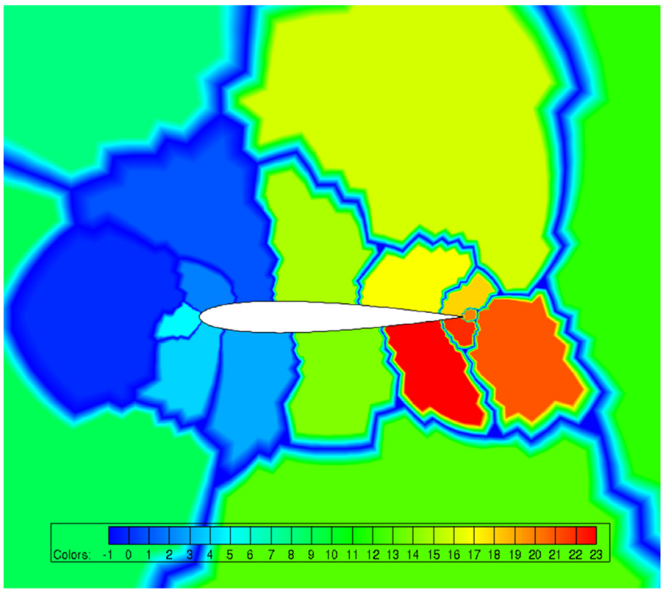
Colors: -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Fig. 7. Baseline performance profile.

A basic load balancing is performed after the METIS decomposition by assigning ownership of repeated edges/vertices in a greedy manner to the thread with the least owned edges/vertices. An example of the edge decomposition for an unstructured mesh around a NACA 0012 airfoil is shown in Fig. 6 with each color representing a sub-domain of edges separated by a dark outline.

5.2. Linear solver optimizations

For the current application, we use the agglomeration multigrid solver for the Navier–Stokes (mean flow) equations, which accounts for the majority of the execution time, as seen in the performance profile of Fig. 7, and an ILU(0)-preconditioned GMRES for the turbulent equation. For the multigrid algorithm, we choose an ILU(0) smoother on each grid level.

As ILU(0) is used for both the multigrid and GMRES methods, the computation pattern remains similar for both solvers. The pattern entails an approximate factorization, backward substitution, and sparse matrix-vector product to compute the residual. These operations fall under the *sparse, narrow-band recurrences* pattern and typically have limited parallelism that is proportional to the number of independent edges of the Jacobian matrix. For a RANS calculation, we solve a coupled system that features

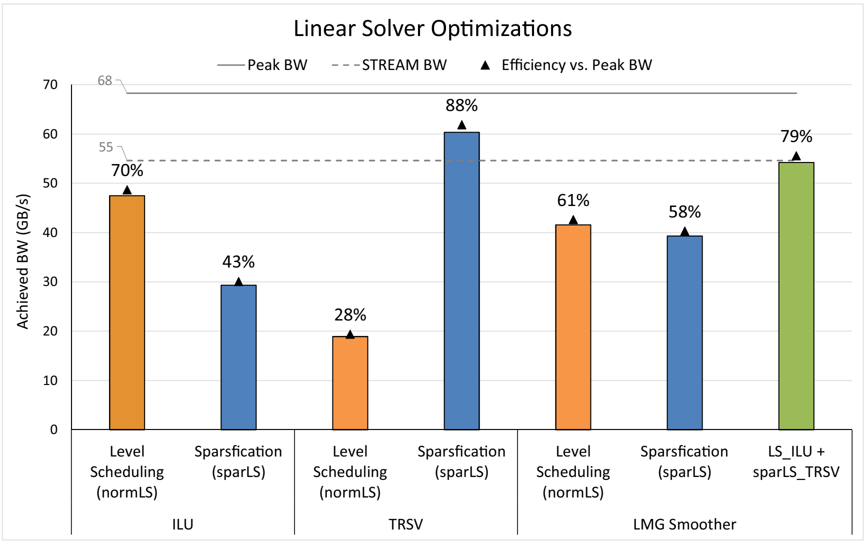

Fig. 8. Linear solver optimizations.

block-structured, diagonally dominant Jacobian matrices. For the mean flow equations, each non-zero element corresponds to a 5x5 block in 3D, while for the scalar turbulence equation, we have a non-block matrix. The decomposition of the Jacobian matrix is done once with each nonlinear (outer) iteration in our steady ONERA M6 tests. However, reusing the factors even as the Jacobian evolves underneath is a problem-dependent optimization that is worth pursuing in practice.

5.2.1. *Threading*

The available parallelism is limited with the recurrences. We pursue a level scheduling-based approach to extract the parallelism for these operations. Due to the different flop/byte ratios for the decomposition and solve phases, we use two variants of the level scheduling that are optimized for each case:

1. Level-scheduling with barriers. We execute task graphs at the granularity of levels with barrier synchronization after each level. The level of a task is defined by the longest path length between the task and an entry of the task dependency graph. As tasks in the same level are independent, they can execute in parallel [33,34]. This is better suited for the decomposition due to the relatively higher compute load found in this phase.
2. Sparsification with approximate transitive edge reduction (P2P-Sparse) by Park et al. [35]. This approach sparsifies (i.e., eliminates) the redundant dependencies with point-to-point synchronization and improves scalability. We use this for the sparse triangular solver, where less compute load is available, to alleviate significant barrier costs.

The performance of these kernels is limited by the memory (DRAM) bandwidth (BW) due to their low flop/byte ratios (i.e., the kernels are bandwidth-bound). Fig. 8 shows the achieved BW relative to machine peak and also relative to the STREAM [36] BW, which represents the maximum achievable BW for streaming accesses (no reuse). In addition, Fig. 8 compares between the above discussed threading strategies and contrasts their benefits for the individual kernels as well as the full smoother function.

The decomposition (ILU) involves a relatively higher compute load, since for each block, the non-zero blocks in that particular row/column prior to it need to be processed [37]. The primary compute is the large number of 5 × 5 matrix inversions, along with matrix–matrix and matrix-vector products. Also, for the incomplete LU decomposition, accessing each non-zero block also accesses the other non-zero blocks in the same column, which results in a highly irregular access pattern. For these reasons, the achieved BW is relatively lower at 70% of peak with level-scheduling with barriers.

The backward substitution phase (TRSV) has far lower compute, as it involves only matrix-vector operations for a non-zero block. This pattern is far more BW-intensive, and the access pattern is more regular with some reuse. Hence, we achieve much higher BW utilization for the TRSV at 88% of peak. The utilization is higher than that of STREAM due to the reuse.

For the full smoother, we achieve very close to STREAM BW at 79% of the peak with this optimal combination of the level scheduling algorithms, as can be seen from Fig. 8. We further observe that the BW starts to saturate beyond 8 cores, which is typical for BW-bound workloads.

5.2.2. *Additional single-node linear solver optimizations*

Here, the core compute operations are the 5 × 5 matrix inversion and matrix–matrix products. Highly-tuned vector implementations are employed for these operations. For the matrix–matrix product, we use an optimized kernel (assembly) generated with libxsmm[3]. The matrix inversion routine is fully unrolled and modified to use static indexing in order to assist auto-vectorization. However, the benefits from these compute optimizations are only realized up to the point when the memory bandwidth is saturated.

Fig. 9 presents the performance benefits with different optimizations both at the kernel level and across the full linear solver. With our optimizations, the ILU kernel achieves a speedup of 3*X*, the sparse triangular solver shows a speed up of 2.2*X*, and the sparse matrix-vector product (residual calculation) improves by 17*X* with 14 cores (14 threads) over the SU2 baseline version using 14 MPI ranks. Overall, speed ups of 2.5*X* and 10.6*X* are achieved for the multigrid and GMRES linear solvers, respectively.

Fig. 9 also compares the performance of the optimized linear solver and kernels (single node optimizations plus threading) against the baseline version on a single core. We also present the baseline version with an MPI rank per core and the single-node

[3] https://github.com/hfp/libxsmm, which creates assembly (small dense) and intrinsic (small sparse) matrix multiplication kernels in single and double precision for the SSE3, AVX, AVX2, MIC and AVX512 x86 instruction set extensions. Up to 90% peak performance is achieved on Intel® Xeon®.

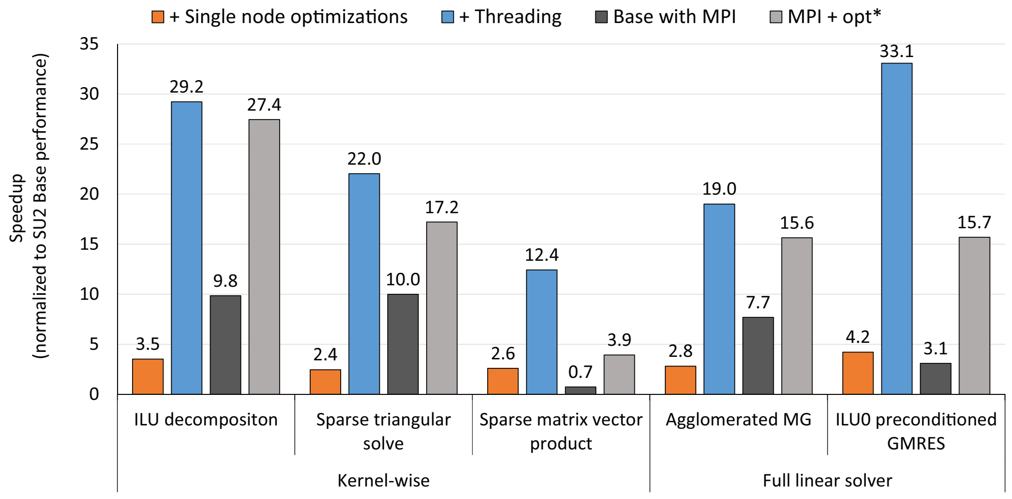

Fig. 9. Linear solver performance.

optimized version with MPI instead of threading, so that the performance with threading versus MPI can be directly addressed.

There are two important things to note about why this is not an entirely fair comparison that also highlight the practical differences in these approaches for this application. Firstly, with increasing rank counts, the individual mesh partitions become smaller, which results in fewer total multigrid levels. This impacts the amount of compute available and the convergence behavior of the algorithm. For the coarse ONERA M6 mesh, the number of multigrid levels is reduced to 3 from the 5 levels achieved on a single rank, as the automatic agglomeration algorithm terminates once an inadequate number of control volumes remain for creating an appropriately-sized coarser level within each partition. This restriction can be avoided if the geometric multigrid algorithm is implemented in a manner that allows agglomeration across the partition boundaries or if the partition boundary locations vary by level. There are other performance issues to consider in this scenario, and it is not pursued in this work. Secondly, execution with a single rank per core introduces additional communication overheads across the ranks even within a single node. The fact that these two issues can be avoided by using threading gives it an advantage over MPI for this application.

Therefore, it can be seen in Fig. 9 that, even though the kernel-wise performance is similar (the optimized and threaded implementation is only slightly better than the optimized code running with a rank per core), there is a significant difference in the overall linear solver performance due to the additional communication costs.

5.3. Edge loop treatment

Edge-based loops predominantly occur in the flux routines, Jacobian matrix construction, gradient calculation, and limiter evaluations. These loops capture the physics of the problem and generate significant computational work per pair of vertices that share an edge. In fact, the majority of the floating-point operations within the application can be found here. Typically, these loops have color-wise concurrency (i.e., edges that do not share the same vertices can be processed in parallel) and require nearest neighbor communication to complete the edges cut by the grid decomposition for MPI. However, coloring-based partitioning of an unstructured mesh results in sub-optimal spatial locality among the concurrently processed edges [38]. To alleviate this issue, we use the edge decomposition strategy discussed at the beginning of this section.

For optimization candidates, we considered three of the most time-consuming edge loops with relatively high computational intensity in terms of flop/byte ratio. These are the loops that compute the convective and viscous fluxes (which also compute their Jacobian entries), limiters, and gradients. The optimization strategy is twofold: firstly, by reducing the memory bandwidth and latency pressure through compacting the working sets and re-ordering the points for efficient cache utilization, and secondly, by optimizing the compute in these loops until the performance for these become as close to bandwidth-bound as possible. Next, we describe the individual optimizations in detail and present the speedups obtained.

5.3.1. Memory optimizations

For the first memory optimization, we apply the Reverse Cuthill-Mckee (RCM) [39] re-ordering technique on the mesh. The mesh edges are re-ordered according to points, i.e., the first edge-point increases monotonically as one progresses through the edges. We also minimize the jump in the second edge-point [40]. This reduces the overall edge adjacency matrix bandwidth [41]. For example, for one of the tetrahedral ONERA M6 meshes, the adjacency matrix bandwidth reduced from 170,691 to 15,515. RCM re-ordering enables a cache efficient traversal of the edge loops and leads to notable performance gains.

Another key memory optimization is an array-of-structures (AOS) to structure-of-arrays (SOA) transformation. In object-oriented fashion, the baseline code is written in AOS form within the main *CSolver* class. More specifically, the *CSolver* class allocates a double pointer to an object of the *CVariable* class that contains the solution variables (unknowns) at a given vertex of the mesh, such as fluid pressure p or the fluid velocity vector (u, v, w), for example. Thus, when accessing these quantities within an edge loop, an indirect access is required to dereference the *CVariable* object at the given vertex. These variables are stored in memory as $[p_1, u_1, v_1, w_1]$, $[p_2, u_2, v_2, w_2]$, ..., where x_i denotes the variable x at point i. The memory addresses for each set of vertex data can be spread out across large distances in memory, which results in expanded working sets. This structure has been modified to the SOA format by removing the *CVariable* class entirely, and the required variables at all of the vertices are stored contiguously as members of the *CSolver* class itself. This avoids the need of indirect access. In the SOA format, the variables are stored in memory as

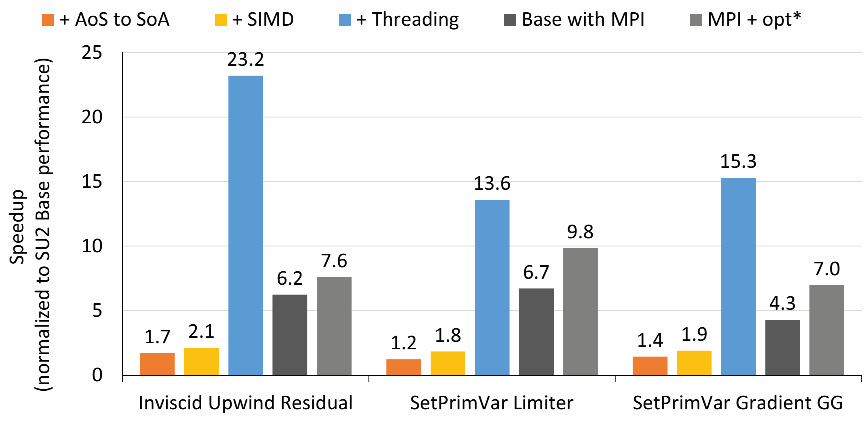

Fig. 10. Edge-loop kernel performance.

$[p_1, u_1, v_1, w_1, p_2, u_2, v_2, w_2, \dots]$, which compacts the working sets and provides cache-efficient traversal of the edge loop.

The speedups obtained from the AOS to SOA transformations are shown in Fig. 10. For the three edge-loops considered, a speedup ranging from 1.2*X* to 1.7*X* is obtained. Lastly, it is important to note that, while the AOS to SOA transformation offers improved performance, it does so at the loss of some flexibility and clarity in the original object-oriented design of the code.

5.3.2. SIMD vectorization

For the residual and limiter routines, we vectorize across the edges. Each SIMD lane processes an edge entirely. Given our hardware, this means that we have four edges being processed concurrently within a thread. To address the possible dependency across these edges, we separate the write and compute phases. The dependency is eliminated for the compute by storing the computed values in a temporary buffer for each SIMD width of edges. After the compute, we use scalar operations to write out results from the temporary buffer. The performance impact from the scalar write-out is minimal, since it is amortized by a large amount of compute in the kernel. The residual edge loops contain inner loops over the variables (such as fluid pressure and velocities, typical loop count = 5 to 7) and also loops over the physical dimensions (loop count = 3 for a three-dimensional mesh). Moreover, they also contain function calls which are converted into “elemental” functions [42].

For the gradient computation, we vectorize across the outer loop over the flow variables, which contains an inner loop over physical dimensions. It is critically important for performance that the scalars and arrays referenced inside the vectorized loops are accessed in a unit-strided fashion with respect to the SIMD-lanes, to avoid any gather and scatter instructions in the vectorized kernels. For this purpose, some data layout modifications are added, especially for the arrays, by promoting them to a higher dimension and switching the inner and outer indexes. We note from Fig. 10 that vectorization increases the speedup over the baseline code by 1.9*X* to 2.1*X* for the three edge-loop kernels.

The effect of OpenMP threading is then considered on top of the vectorization for these kernels. For 14 cores running 14 threads, we get a speedup of 3.7*X*, 2.0*X*, and 3.5*X* for the residual, limiter, and gradient kernels, respectively, over the baseline code with 14 MPI ranks, as seen in Fig. 10. In this figure also the speedups are shown relative to the single core baseline. As can be seen from this figure, scaling the optimized code on 14-cores using threading is much more performant than scaling it with MPI. The performance advantage of threading over MPI is 1.39*X*, 2.19*X*, and 3.05*X*, for limiter, gradient, and residual kernels, respectively.

6. Distributed memory, coarse-grained optimizations

As is typical of most modern CFD codes, SU2 was originally designed with distributed memory computing through MPI in mind. SU2 follows a standard approach for achieving its coarse-grained parallelism: the computational mesh is partitioned into domains that are assigned to each rank, and the code executes in single process, multiple data (SPMD) mode with a number of synchronization points during each iteration when data is communicated to nearest neighbors or possibly across all ranks with collective communications.

More specifically, the mesh partitioning process is the first task upon launching SU2, and this is accomplished entirely in parallel by providing a simple linear partitioning of the mesh to the ParMETIS software [43], which then provides higher quality partitions in terms of load balance and minimized edge cut. The partitions are then post-processed in order to locate neighboring (adjacent) partitions, to create a single halo layer around each partition that overlaps with these neighbors (replicated layer of halo cells and ghost vertices), and to define a communication schedule for when nearest neighbor communications are necessary.

Nearest neighbor communications of data are completed using non-blocking `MPI_Isend()` and `MPI_Irecv()` calls. Each rank cycles through its list of nearest neighbors and posts all non-blocking send and receive calls before accepting messages in a first-come, first-served order with `MPI_Waitany()`. A number of collective communications are typically used to monitor the convergence of the flow solver with each nonlinear iteration by tracking global residuals or force coefficients on the aerodynamic surfaces. However, nearly all of these communications can be avoided by checking for convergence less frequently or not at all (e.g., running a fixed number of iterations), and this does not affect solution correctness. Care has been taken to avoid these extraneous communications in the results that follow, and it should be noted that the avoidance of these global synchronization points has a noticeable impact on scalability.

With each nonlinear iteration of the RANS equations in SU2, several critical pieces of data must be communicated with nearest neighbors in order to complete the residual and Jacobian calculations in control volumes along the partition boundaries. For our particular set of chosen numerical methods, the gradients of the flow variables and the limiter values must be communicated to neighbors before each partition can compute the residuals through the series of edge loops. With multigrid, these communications should occur on each coarse mesh level as well.

The residuals and the Jacobians are then passed to either GMRES or multigrid, where a number of additional communications

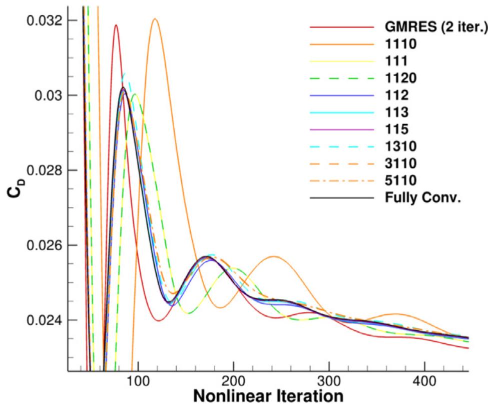

Fig. 11. Coefficient of drag convergence behavior for different linear solver configurations.

are required during the linear solve. For multigrid, a single nearest neighbor communication of the solution is necessary after performing each ILU(0) smoothing iteration on each level of a multigrid cycle. For GMRES, two nearest neighbor communications are performed per iteration (ILU(0) preconditioner and matrix-vector product) along with additional global reductions during dot product operations. Previous work has established that 2 GMRES iterations per nonlinear iteration of SU2 offers an ideal balance between work per iteration and nonlinear solution convergence to a steady state [44].

After completing the linear solve, a final solution update occurs, including one more nearest neighbor communication of the new solution before starting the next nonlinear iteration. Note that these communication requirements apply to both the mean flow and turbulent equations, with the exception that limiters are not necessary for the turbulence equation when using a first-order convective scheme.

7. Performance results

In this section, we describe the single- as well as multi-node performance results for the full SU2 application based on RANS calculations with the ONERA M6 test case.

7.1. *Establishing a linear solver configuration*

In order to establish fair comparisons against GMRES and to better understand the ideal configuration, we first perform an exploration of the available settings for the LMG solver with the coarse ONERA M6 case on a single node and a single MPI rank. As noted above, the automatic agglomeration procedure will continue creating coarse grid levels until a prescribed threshold is reached where an insufficient number of control volumes remain for agglomeration. The LMG solver is capable of executing multiple V-cycles at each nonlinear iteration, and it can also apply variable numbers of pre- and post-smoothing iterations at each grid level during a single v-cycle.

The following LMG parameter study compares the convergence behavior of LMG under many scenarios against that of GMRES (with 2 linear iterations per nonlinear iteration) and the convergence behavior when the linear system is converged to machine tolerance with each nonlinear iteration. For all cases, a CFL number of 20 was chosen, which was the largest stable value for both GMRES and LMG on this particular mesh and set of flow conditions. The convergence of the coefficient of drag is used as the metric of interest.

Fig. 11 contains the history of the drag coefficient for various sets of LMG parameters. The numerical key in the legend represents the configuration in terms of number of grid levels, number of V-cycles, and number of smoothing iterations. For example, “335” represents a configuration with 3 grid levels for the multigrid algorithm, 3 V-cycles per nonlinear iteration, and 5 pre- and post-smoothing ILU(0) iterations on each level, respectively. An additional “0” in the final entry signifies that the post-smoothing was turned off, leaving only pre-smoothing iterations during the V-cycle.

Given the results in Fig. 11, we can draw the following conclusions from the parameter study:

1. The 1110 configuration is the farthest from the fully-converged linear solve, as would be expected with only one pre-smoothing iteration. Here, we are not technically applying multigrid, rather we are simply applying one smoothing step to the system as would be done with a classical iterative smoother in a single grid algorithm.
2. Once a second smoothing step without multigrid is added, either as a pre- (1120) or post-smoothing (111), the convergence behavior approaches that of the fully-converged system. Note that configurations 1120 and 111 should exhibit identical results, as we are again only operating on the fine grid without multigrid.
3. As we continue adding smoothing iterations without multigrid, the convergence improves, as would be expected. Configuration 1310 demonstrates a situation that is equivalent to 3 smoothing iterations on the fine grid. This suggests that the smoother applied to the fine grid alone with an increased number of smoothing iterations (e.g., between 2 and 4) without multigrid may be worth exploring at scale. However, anything beyond 4 smoothing iterations (112, 113, or 115) provides diminishing returns in terms of nonlinear convergence.
4. Configurations 1110, 3110, and 5110 represent configurations with 1, 3, or 5 levels of multigrid, respectively. 3110 and 5110 provide very similar results that are very near the fully converged system behavior, suggesting that using 3 levels or more of multigrid with only a single pre-smoothing iteration offers effective convergence without requiring multiple smoothing iterations at each grid level.

Given the promising results for using multigrid with a single pre-smoothing iteration, we choose this LMG configuration for the remainder of the experiments in this article. This configuration provides effective convergence behavior while reducing the computational effort and communication requirements for each nonlinear iteration. For the GMRES cases, we will continue using 2 iterations per nonlinear iteration, which was shown in previous work to be a good compromise between convergence and computational efficiency [44].

7.2. *Single-node results*

For the single-node experiments, we use the coarse ONERA M6 mesh, which is representative of the typical RANS problem size that would be executed on a single compute node. Our optimized version of SU2, with 14 threads on 14 cores, exhibits a 15.1*X* speedup over the baseline version running on a single core and a 2.6*X* speedup over the baseline version running with 14 MPI ranks.

Fig. 12 shows the performance benefits for each computational pattern normalized to the baseline SU2 performance on a single core and also compares with the baseline version running on 14 MPI ranks within a socket. Here, the linear solver combines both the agglomeration multigrid for the mean flow and preconditioned GMRES for the turbulence equation.

We can now dissect the overall speedup for the full application and quantify the benefits from the different stages of optimizations, which is shown in Fig. 13 as accumulated speedups for

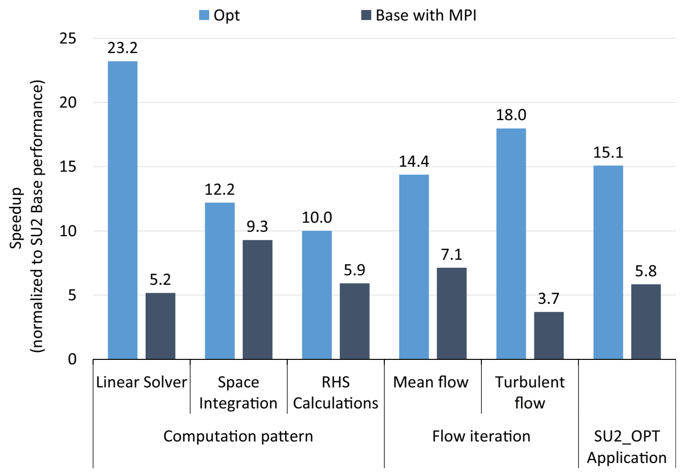

Fig. 12. SU2 optimized performance speedup.

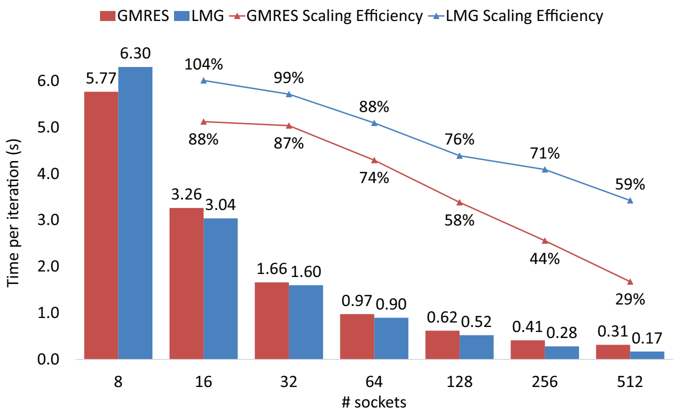

Fig. 14. Scaling the ONERA M6 fine mesh to 256 nodes.

each stage. Furthermore, we compare the results to the scenario with MPI-only parallelization across cores. The 2.6*X* speedup over the baseline version running 14 MPI ranks can be further divided, with 2.2*X* coming from the single-node optimizations (SIMD, linear solver, and data structure optimizations) and the remaining 1.2*X* from using threads. This clearly indicates that single-node optimizations are more important than threading alone for this application. Again, we note that flat MPI and multi-threading can not be directly compared (the increased MPI ranks impose other challenges), so this should be considered a qualitative assessment.

Our characterization indicates that the overall application is expected to be bound by the memory bandwidth, since the flop/byte ratio is quite low. Hence, in going from a single core to 14 cores, we see only a 7.5*X* improvement, which is the typical scaling pattern for memory bandwidth-bound workloads. For the full application on a single 14-core Xeon E5-2697 v3 socket, we achieve very close to the STREAM bandwidth. After optimizations, we see a flatter performance profile with the linear solver, space integration kernels, and right-hand side calculations each equally contributing ($\approx$ 30%) to the total execution time, with 80% of the total time still spent in the Navier–Stokes (mean flow) portion.

It is important to highlight that, since most of our optimizations are targeted towards the critical compute kernels of SU2 (i.e., the linear solver, edge loops, residuals, gradients, and limiters), they can be easily adapted to other applications. Therefore, our results here are extensible to other similar CFD applications for exploiting the potential of current and upcoming highly parallel computer architectures.

7.3. Multi-node results

Having shown that our optimized implementation greatly improves the performance of SU2 in a single-node context, we now demonstrate its scalability on multiple nodes. Multi-node scalability is essential for solving large-scale, real-world applications, such as the flow around complete aircraft configurations. With the multi-node results, we will showcase the following:

1. Superior scalability of the agglomeration multigrid-based linear solver over the conventional GMRES linear solver (Krylov-based methods).
2. Improved strong scaling potential with hybrid parallelism (OpenMP + MPI).

To demonstrate this, we perform strong scaling studies with the fine ONERA M6 mesh on up to 256 nodes of the Endeavor cluster and compare the performance for the LMG and GMRES linear solvers running with the optimized version of the code (including hybridization), as shown in Fig. 14. We use 512 MPI ranks with 2 ranks per node and 7168 total cores. In the scaling experiments, we use a CFL of 1 for both the agglomeration multigrid and GMRES

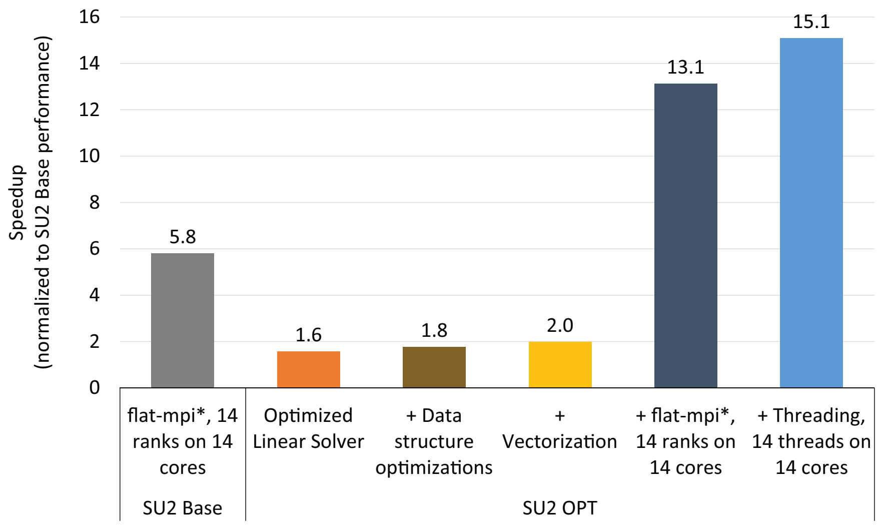

Fig. 13. Full application performance breakdown.

tests and compare time per iteration. With a fixed CFL, both solvers will have similar nonlinear convergence behavior and hence require a similar overall number of nonlinear iterations for full solution convergence (outer loop). The study concerning our solver configuration at the beginning of this section, which established that the chosen LMG solver configuration approaches the convergence of the fully-converged linear system and GMRES with each nonlinear iteration, ensures that the following comparisons of time per nonlinear iteration are meaningful in the context of the complete solution process.

The memory requirements of the ONERA M6 fine mesh require that we run on a minimum for 4 nodes (8 ranks), which serves as the baseline case. We can see from Fig. 14 that, for the smaller node count tests, the time per iteration for multigrid is higher, which is due to the increased computation within the multigrid algorithm. However, as we push to scale, there is a crossover point where multigrid outperforms GMRES. The improved scaling for multigrid is derived from the fact that only nearest neighbor communications are required, whereas GMRES, and Krylov-based methods in general, are limited by the costly global reduction operations [7,38]. At 256 nodes with multigrid, we see a 1.9*X* speedup in time per iteration.

Executing the hybrid implementation (OpenMP + MPI) with the LMG solver at 256 nodes on the fine ONERA M6 mesh (9.25*e*6 points), we achieve 59% scaling efficiency with approximately 1.3*e*3 points per core. Whereas with a flat-MPI scenario (single MPI rank per core), we can not run beyond 64 nodes (1792 MPI ranks), due to very small coarse-grained partitions. This indicates that the performance stops scaling at approximately 5*e*3 points per core. With the hybrid implementation on the current Haswell family multicore Xeon CPU (14-core E5-2697 v3) cluster, we have been able to push the strong scaling limit by 3.8*x*. This can only be expected to improve with increased parallelism within a node.

8. Conclusions

In this paper, we have presented our optimizations of a full CFD application for scalable RANS calculations on unstructured meshes. The code is based on the widely-used, open-source SU2 package. Our holistic combination both single- and multi-node optimizations, as well as the use of a scalable linear multigrid solver, has enabled our code to exhibit a near 4*X* improvement in strong scaling on 256 compute nodes, as compared to the baseline implementation using a GMRES linear solver and a flat-MPI configuration.

Applying the agglomeration multigrid solver resulted in close to a 2*X* speedup on 256 nodes, as compared to the baseline GMRES linear solver. This performance boost is due to a significant reduction in `MPI_Allreduce()` operations, which are known to be major limiters to the scalability of Krylov solvers, such as GMRES. In contrast, linear multigrid offers the desired computational complexity and better parallel scalability due to the reduced amount of global reductions. Given the trend of increasing hardware parallelism, this makes it an ideal choice for emerging, extreme-scale systems, and thus, it inspires future research for these algorithms in the context of CFD.

While the modern x86 microarchitecture is designed for high single-thread performance, we observe significant benefits with the single-threaded code optimizations: SIMD vectorization, data layout, edge reordering, and conflict avoidance within SIMD lanes resulted in 2.2x speedup, as compared to the baseline single-threaded performance. An interesting future research direction is to expose some of these optimizations to the compiler, as modern compilers are not able vectorize loops with potential conflicts. However, upcoming cross-lane conflict detection in AVX3 may enable automatic vectorization of such loops.

We have also observed a speedup due to parallel shared memory optimizations. Namely, while using the same single-thread optimizations, our single-node OpenMP version runs 1.2*x* faster on 14 cores than a flat-MPI implementation on the same number of cores. This suggests that, while a flat-MPI programming model is simpler to implement, a hybrid parallel programming model that combines OpenMP within a node and MPI across nodes has an advantage for this type of CFD application. Additional benefits may be realized by using fewer MPI ranks, such as higher quality (larger) partitions and both higher parallel efficiency and faster convergence with the multigrid algorithm due to improved agglomeration of the larger partitions.

Finally, as all of our optimizations have been implemented within the context of the open-source SU2 suite, we would like to underscore that this work will be beneficial to the greater CFD community.

Acknowledgments

The authors would like to thank Mallick Arigapudi and Michael Hebenstreit of the Intel Corporation for their help with running the jobs on the Endeavor cluster. The authors would also like to thank Travis Carrigan of Pointwise, Inc. for sharing high quality ONERA M6 meshes for our numerical experiments. T. Economon, F. Palacios, and J. Alonso would like to acknowledge the Intel Corporation for their support of the Stanford University Intel Parallel Computing Center.

References

[1] Jameson A. The present status, challenges, and future developments in computational fluid dynamics. In: AGARD, editor. Progress and challenges in CFD methods and algorithms.
[2] Vassberg JC, DeHaan MA, Rivers SM, Wahls RA. Development of a common research model for applied cfd validation studies. AIAA2008;(2008-6919).
[3] Gropp WD, Kaushik DK, Keyes DE, Smith BF.
[4] Aubry R, Houzeaux G, Vasquez M. Some useful strategies for unstructured edge-based solvers on shared memory machines. AIAA Paper 2011-06142011.
[5] Mavriplis DJ, Mani K. Unstructured mesh solution techniques using the NSU3D solver
[6] Duffy A, Hammond D, Nielsen E. Production level CFD code acceleration for hybrid many-core architectures. NASA/TM-2012-217770, 2012.
[7] Mudigere D, Sridharan S, Deshpande A, Park J, Heinecke A, Smelyanskiy M, et al. Exploring shared-memory optimizations for an unstructured mesh cfd application on modern parallel systems. In: Parallel & Distributed Processing Symposium (IPDPS). IEEE; 2015.
[8] Economon TD, Palacios F, Copeland SR, Lukaczyk TW, Alonso JJ. Su2: an open-source suite for multi-physics simulation and design. AIAA Journal 2015. doi:10.2514/1.J053813.
[9] Hirsch C. Numerical Computation of Internal and External Flows. New York: Wiley; 1984.
[10] Wilcox D. Turbulence Modeling for CFD. 2nd ed. DCW Industries, Inc.; 1998.
[11] White F. Viscous Fluid Flow. New York: McGraw Hill Inc.; 1974.
[12] Spalart P, Allmaras S. A one-equation turbulence model for aerodynamic flows. AIAA Paper 1992-04391992.
[13] Barth TJ. 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; Von karman Institute for fluid dynamics, Rhode Saint Genese Begium; 1995.
[14] Quarteroni A, Valli A. Numerical approximation of partial differential equations. Springer series in computational mathematics, vol. 23. Springer-Verlag Berlin Heidelberg New York; 1997.
[15] Jameson A. A perspective on computational algorithms for aerodynamic analysis and design. Progr Aerosp Sci 2001;37:197–243.
[16] LeVeque R. Finite Volume Methods for Hyperbolic Problems. Cambridge Univesity Press; 2002.
[17] Wesseling P. Principles of Computational Fluid Dynamics. Springer Series in Computational Mathematics, vol. 29. Springer-Verlag Berlin Heidelberg New York; 2000.
[18] Jameson A. Analysis and design of numerical schemes for gas dynamics, 1: artificial diffusion, upwind biasing, limiters and their effect on accuracy and multigrid convergence. Int J Comput Fluid Dyn 1995;4(3–4):171–218. doi:10.1080/10618569508904524.
[19] Jameson A. Analysis and design of numerical schemes for gas dynamics, 2: artificial diffusion and discrete shock structure. Int J Comput Fluid Dyn 1995;5(1–2):1–38. doi:10.1080/10618569508940734.
[20] Toro EF. Riemann Solvers and Numerical Methods for Fluid Dynamics: a Practical Introduction. Springer-Verlag; 1999.

[21] Roe PL. Approximate Riemann solvers, parameter vectors, and difference schemes. J Comput Phys 1981;43:357–72.
[22] van Leer B. Towards the ultimate conservative difference scheme V. a second-order sequel to Godunov's method. J Comput Phys 1979;32(1):101–36.
[23] Venkatakrishnan V. On the accuracy of limiters and convergence to steady state solutions. AIAA Paper 1993-08801993;.
[24] Blazek J. Computational Fluid Dynamics: Principles and Applications. Oxford: Elsevier; 2005.
[25] Eliasson P. Edge, a Navier-Stokes solver for unstructured grids. Tech. Rep. FOI-R-0298-SE FOI Scientific Report; 2002.
[26] Economon TD, Palacios F, Alonso JJ, Bansal G, Mudigere D, Deshpande A. et al. Towards high-performance optimizations of the unstructured open-source su2 suite. AIAA Paper 2015-1949.
[27] Saad Y, Schultz MH. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J Sci Stat Comput 1986;7:856–69.
[28] Gropp WD, Kaushik DK, Keyes DE, Smith BF. High-performance parallel implicit CFD. Parallel Comput 2001;27(4):337–62.
[29] Gropp WD, Kaushik DK, Keyes DE, Smith BF. Performance modeling and tuning of an unstructured mesh CFD application. In: Supercomputing, ACM/IEEE 2000 Conference. IEEE; 2000.34–34
[30] Schmitt V, Charpin F. Pressure distributions on the ONERA-M6-wing at transonic mach numbers. Tech. Rep.; 1979. AGARD, Report AR-138.
[31] Hammarlund P, Kumar R, Osborne RB, Rajwar R, Singhal R, D'Sa R, et al. Haswell: the fourth-generation intel core processor. IEEE Micro 2014;34(2):6–20.
[32] Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 1998;20(1):359–92.
[33] Anderson E, Saad Y. Solving sparse triangular linear systems on parallel computers. Int J High Speed Comput 1989;1(01):73–95.
[34] Naumov M. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Tech. Rep. NVIDIA Technical Report, NVR-2011-001; 2011.
[35] Park J, Smelyanskiy M, Dubey P. Sparsifying synchronizations for high-performance shared-memory sparse triangular solver. In: International Supercomputing Conference (ISC); 2014.
[36] McCalpin J. Stream benchmark. www.cs.virginia.edu/stream/ref.html#what 1995;.
[37] Chow E, Saad Y. Experimental study of ILU preconditioners for indefinite matrices. J Comput Appl Math 1997;86(2):387–414.
[38] Mavriplis D, Mani K. Unstructured mesh solution techniques using the NSU3D solver. In: AIAA Paper 2014-0081, Presented at the 52nd AIAA Aerospace Sciences Conference, National Harbor, MD; 2014.
[39] Cuthill E, McKee J. Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th national conference. ACM; 1969. p. 157–72.
[40] Lohner R. Cache-efficient renumbering for vectorization. Int J Numer Method Biomed Eng 2010;26(5):628–36.
[41] Duff IS, Erisman AM, Reid JK. Direct Methods for Sparse Matrices. Monographs on Numerical Analysis. Oxford University Press; 1989.
[42] Intel® architecture instruction set extensions programming reference 2014.
[43] Karypis G, Schloegel K, Kumar V. Parmetis: Parallel graph partitioning and sparse matrix ordering library. Version 10 Department of Computer Science, University of Minnesota; 1997.
[44] Palacios F, Economon TD, Aranake AC, Copeland SR, Lonkar AK, Lukaczyk TW. et al. Stanford University Unstructured (SU^2): Open-source analysis and design technology for turbulent flows. AIAA Paper 2014-02432014.