Sustainable High-Performance Optimizations in SU2

Conference Paper · January 2021

DOI: 10.2514/6.2021-0855

CITATIONS
7

READS
799

3 authors:

Pedro Gomes
Imperial College London
8 PUBLICATIONS **50** CITATIONS
SEE PROFILE

Thomas D. Economon
Stealth-Mode Tech Startup
59 PUBLICATIONS **2,139** CITATIONS
SEE PROFILE

Rafael Palacios
Imperial College London
180 PUBLICATIONS **2,734** CITATIONS
SEE PROFILE

Sustainable High-Performance Optimizations in SU2

Pedro Gomes*
Imperial College London, London, SW7 2AZ, United Kingdom

Thomas D. Economon†
SU2 Foundation, San Francisco, CA, 94158, U.S.A.

Rafael Palacios‡
Imperial College London, London, SW7 2AZ, United Kingdom

Over a period of approximately 18 months, we have achieved an average 4-fold performance increase of the open source multiphysics suite SU2 through implementation optimizations (e.g. vectorization), and for some problems an additional 10-fold improvement via algorithmic changes. We have implemented a hybrid parallelization strategy (MPI + OpenMP) that improves the scalability of the code and allows key algorithms (such as multigrid) to maintain their effectiveness at small number of nodes per core. Our work has maintained the generality and versatility of the code by not relying on optimizations specific to given compilers, architectures, or physics. Furthermore, we maintain, or lower, the level of C++ knowledge needed for new developers. In this paper we document the implementation and algorithmic changes, give an overview of the details that allow implementing the hybrid parallel and vectorization frameworks in a way that hides the low-level complexity from high-level algorithm development. We demonstrate the improvements on benchmark problems known to the aeronautics community, and derive best practice guidelines to use the new capabilities.

I. Introduction

The hierarchy of classes used in SU2 has been previously described[1], notwithstanding, it is worth presenting a summarized version of the classes that were targeted by this work. The *Solver* hierarchy defines solvers for particular PDE problems, the key roles of a solver are to define the pre processing operations required before edge fluxes (and volumetric sources) can be computed, orchestrate the computation of those fluxes, and implement boundary conditions. Spatial discretization methods (JST, Roe, etc.) are implemented by lightweight *Numerics* classes, which are *visitor* classes that solvers use to compute fluxes and source terms. Solution variables are stored in *Variable* classes and usually a *Solver* has an associated *Variable*. In summary, a solver has variables and uses numerics, as shown in Fig. 1.

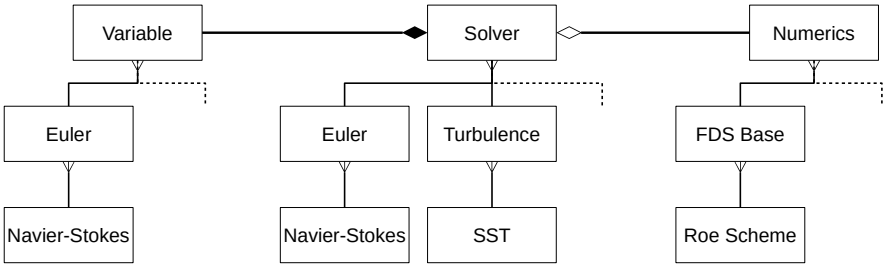


Fig. 1 Simplified UML diagram of the main SU2 classes targeted for optimizations.

In addition to those, geometric information associated with the grid (coordinates, normal vectors, adjacency matrices, etc.) is stored and manipulated by *Geometry* classes that all solvers use. All these classes are polymorphic, which makes possible defining higher level algorithmic steps, e.g. spatial and temporal integration, applicable to a range of

*PhD Student, Department of Aeronautics, p.carrusca-gomes18@imperial.ac.uk.
†Executive Director, AIAA Senior Member.
‡Professor of Computational Aeroelasticity, Department of Aeronautics, AIAA Associate Fellow.

problems. Furthermore, inheritance is used to avoid duplication, for example specific turbulence solvers (SA, SST, etc.) are derived from an intermediate turbulence solver class.

The remainder of this paper is organized as follows, section II deals with refactoring some kernel areas of the code to suppress known bottlenecks and facilitate the other developments. Sections III and IV present the new hybrid parallel and vectorization strategies, respectively. The former aims to improve scalability by reducing domain decomposition overheads, while the latter allows modern hardware to be fully utilized without increasing the complexity of high level algorithm development. In section V we describe algorithmic improvements for robust nonlinear convergence. Finally in section VI we show numerical examples ("before/after" comparisons accompany their respective sections).

II. Implementation Optimizations

A. Distributed Memory Enhancements

SU2 follows a standard approach for achieving coarse-grained parallelism in a CFD solver through the use of the Message Passing Interface (MPI). First, the computational mesh is partitioned into subdomains that are assigned to each MPI rank as a preprocessing step, and then 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 (i.e., the ranks sharing subdomain boundaries) or possibly across all ranks with collective communications.

Mesh reading and partitioning are critical first steps of the solution process. In order to treat increasingly large problem sizes, bottlenecks of both compute and memory usage must be avoided. Therefore, all steps of the mesh reading and partitioning should operate in parallel and at no point in the process should a single MPI rank load the entire set of mesh points or elements (other than in the context of a serial calculation on a single rank). At the time of writing, a typical compute node configuration might contain dual socket CPUs with 2-4 GB/core of RAM. If each rank on a node attempts to read the entire mesh in flat-MPI mode (1 rank per core), the 2-4 GB of memory would quickly be exhausted for large meshes (e.g., greater than 100 million cells). Therefore, the typical approach is that each rank reads only a portion of the mesh into memory in a simple linear partitioning before communicating the nodes and elements to their owning ranks.

SU2 performs parallel reads of both a native ASCII mesh format as well as binary CGNS files. For large problems, binary formats allow much more efficient access to the data and have the benefit of smaller size. As an open-source project, it is important to reduce dependencies where possible for ease of use, and therefore, we support CGNS files in ADF format and rely on the serial version of the CGNS library, which can be easily integrated with the SU2 source. In earlier versions of the serial CGNS library, it was not possible to perform a linear partitioning of the mesh cells without reading an entire section into memory, which can exhaust resources as discussed above. However, starting with CGNS v3.4.0, partial reads allow for efficient linear partitioning. The CGNS reader in SU2 was recently rewritten to take advantage of this feature and it is now possible to process CGNS grids in ADF format that contain over 1 billion cells.

After reading the mesh, the ParMETIS software [2] provides higher quality partitions in terms of load balance and minimized edge cut using the initial linear partitioning of the mesh as input. The static load balancing metric (i.e. the estimated work per vertex) is a linear equation ($mx + b$) in the number of neighbors per vertex (x), the weights (m and b) can be tuned to produce either a balanced number of edges per partition (high m, low b), or a balanced number of vertices per partition ($m = 0$). 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()`. The structure of the nearest-neighbor communications was recently refactored and updated, resulting in a more efficient implementation in fewer lines of code.

Simple graph partitioning does not account for the heterogeneous nature of distributed parallel architectures, e.g. communication between two cores on the same CPU is cheaper (higher bandwidth and lower latency) than between two machines, even if connected by a high performance network. This can significantly reduce the parallel efficiency of the code. A possible solution is to use graph *mapping* strategies where the target architecture is represented as a weighted graph to reflect the different communication costs. However, this approach still implies static load balancing and other sources of inefficiency associated with domain decomposition, like redundant flux computations for cut edges. Therefore, we pursued a hybrid parallel strategy, detailed in section III, that addresses both of these sources of inefficiency.

B. Data Layout Refactoring

Until version 7, the storage layout in SU2 could be described as *arrays of classes*, i.e. solvers would use arrays of *variables*, and all data associated with a node would be close together in memory. As the code grew, and acquired multiphysics capabilities, this architecture became a performance bottleneck. As virtual functions were added to the base class to allow access to particular variables of a new physics problem (or simply to allow common solution variables to be accessed by descriptive names, instead of being treated abstractly) the spatial locality worsened due to the increased distance in memory between the variables of consecutive nodes, noting that virtual functions take up space.

The storage layout introduced in version 7 can be described as *class of arrays of structures* (or row-major matrices), i.e. multiple arrays of structures (for example for the conservative and primitive variables) are encapsulated in *Variable* classes. This higher level encapsulation was kept to allow fine tuning of the layout. For example, if it proves beneficial for some method to store the conservative and primitive variables of each node contiguously, few changes would be required. The encapsulation also allows some variables to be stored only at particular nodes. Row-major storage was kept since the variables of a node (e.g. primitive variables) are passed by pointer to many routines throughout the code, the implications on vectorization are discussed in section IV. In hindsight using *iterator* types would have been more versatile, and allowed the storage layout to change without affecting the algorithms. Separability between algorithms and data structures should be a requirement in any C++ project, *encapsulation* and *generic programming* are the ways to achieve it without sacrificing performance, as shown in literature [3].

The *Variable* classes that encapsulate the storage arrays remain polymorphic, in general virtual functions cannot be *inlined* which introduces significant overhead in bandwidth-bound routines. We remove this overhead in two ways. First, as there is a nearly one-to-one mapping between solvers and variables, the former access their own variables via references (or pointers) to derived instead of base class, this allows compilers to de-virtualize. Second, the arrays of structures stored by the variable classes are of generic types with the same interface, important routines (e.g. computing gradients) were made generic (templated) and operate on the entire data sets (e.g. on the entire field of primitive variables), thereby making any virtual cost $O(1)$ instead of $O(N)$.

Using this generic storage type for solution and geometric variables is also key to implement vectorization, as detailed in section IV, as it centralizes aligned allocation, padding, and high-level vectorized accesses (i.e. retrieving/storing vector-sized variables from/to the container).

C. Loop Fusion

Memory bandwidth is the main hardware bottleneck to low-order finite volume methods (FVM), when considering the typical ratio of FLOPS to MB/s of modern machines, and even before taking advantage of their SIMD capabilities. The single largest block of memory allocated by the solvers is the sparse matrix used to store the residual Jacobian for implicit solution methods. This matrix is populated during edge loops where the numerical classes are used to compute the flux and its Jacobians with respect to the solution at the end nodes of the edge. In previous versions of the code two edge loops would be performed for a viscous problem, one to account for the convective fluxes, and another for the diffusive fluxes. This separation (between Euler and Navier-Stokes solvers and numerics) is physically logical but makes poor use of the available bandwidth. To avoid this overhead the two loops were fused at the numerics level, i.e. we allowed the numerics objects to be *decorated* (at compilation) such that one can define, for example, an optimized numerics class that computes the Roe flux and also the viscous contribution. We also note that with this we sacrifice the ability to (cleanly) implement hybrid multistage explicit time marching schemes.

D. Mixed Floating Point Precision

As mentioned above, the sparse matrix needed for implicit solution methods is the largest chunk of memory allocated by the code. Albeit easy to vectorize, sparse matrix-vector multiplication is a bandwidth-bound operation, since these linear systems only need to be smoothed, we store the matrix in single precision while all other operations in the code are carried in double precision. This improves the FLOP/BYTE ratio of sparse operations without impacting the overall accuracy of the code for problems that are solved iteratively, nevertheless by default the code will be compiled for double precision linear algebra. The impact of this optimization is more significant for centered convective schemes and stretched grids, as both increase the condition number of the Jacobian matrix.

E. Augmented Sparse Storage Format

A block compressed row-major storage format (BCSR) is used for sparse matrices in SU2. In the aforementioned residual loops the matrix is populated by setting these blocks. We avoid having to search for the location of blocks in 1D storage space, by mapping off-diagonal entries to edges and diagonal entries to nodes. For viscous 3D problems this increases the storage requirements of the matrices by approximately 7%. Storing the location of the diagonal entries is also beneficial when computing and applying linear preconditioners, like ILU or LU-SGS, that operate on the lower/upper parts of the matrix in different phases. Furthermore, operations on the blocks of the matrix are accelerated using Intel MKL's just-in-time code generation for GEMM-type operations (matrix-matrix or matrix-vector multiplication).

F. Numerical Demonstration on Benchmark Problem

To study the effect of the above optimizations on performance, we consider first the subsonic flow, Mach 0.6 at 2 degrees angle of attack (AoA), over the wing geometry of Fig. 2, obtained by *lofting* two NACA0012 profiles. The chord is 0.25 m at the root and 0.175 m at the tip, the span is 1 m. The 0.25c line is swept back 5 degrees and the wing has no twist. Convective fluxes are computed with a second order Roe scheme, the gradients via the Green-Gauss theorem, and the flow variable reconstruction limited with Venkatakrishnan and Wang's limiter. Menter's SST turbulence model is used without wall functions. A coarse mesh (approximately 530 000 hexahedrons) is used for these initial investigations as it allows us to easily profile the code, the mesh metrics of this problem are representative of the finer grids used in section VI.

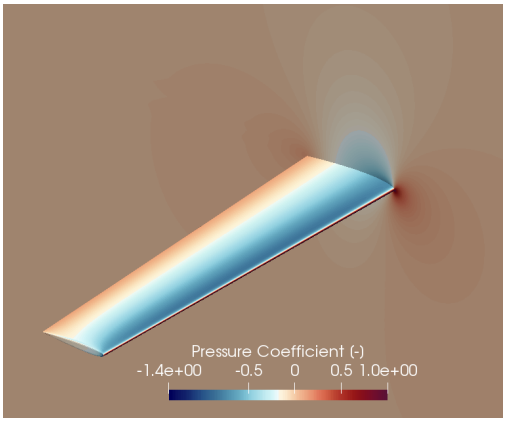


Fig. 2 Geometry, with pressure coefficient contours, of the strong scaling benchmark case.

Solution times (5 orders of magnitude residual reduction) were obtained on a machine with 24 cores (two Intel Xeon E5-2650v4 CPU) with SU2 versions 6.2.0 and 7.0.6. Both versions were compiled with GCC 8.2.0 with optimization flags "-O2 -funroll-loops -ffast-math -march=broadwell -mtune=broadwell"*, Intel MPI 2019.6 and MKL 2019.4 were used (the latter is not relevant to SU2 version 6.2.0). Single core results were not obtained as they are not representative of how CFD codes are used, nor of the compute-to-bandwidth ratio of current mainstream hardware. Table 1 shows wall-time, total number of iterations, and average time per iteration, version 7.0.6 is 3.2 times faster per iteration.

Version	Wall time	Iterations	Time per iteration
6.2.0	776 s	472	1.64 s
7.0.6	237 s	466	0.509 s

Table 1 Comparison of solution times before and after the implementation optimizations.

*The O3 level of optimization reduces the performance of the code, fast-math helps in the vectorization of transcendental operations and dot products, in practice not all optimizations included with it are required. In our experience accuracy is not compromised with GCC, while some of the fast-math optimizations are used by default by the Intel compilers.

III. Hybrid Parallel Strategy

A. Implementation Objectives and Challenges

Domain decomposition is the traditional approach to parallelize CFD codes, with MPI being used to communicate variables at so called halo nodes. Besides the overhead of copying data into (and from) communication buffers, some inefficiency is also incurred by the duplication of some computations at the halo nodes and cut edges. Other downsides of domain decomposition include the difficulty to statically balance computational loads which causes parallel inefficiency when the subdomains are small, and the inefficiency of algorithms that require, or benefit from, un-partitioned domain information (wall distance calculation, geometric multigrid, data interpolation for multiphysics simulations, to name a few). To address them a hybrid strategy has been adopted, that is based on distributing point, edge, and element loops over OpenMP threads, thereby reducing the number of subdomains for the same number of cores, and enabling automatic load balancing within subdomains.

The key challenge of such *shared-memory* parallelization approaches is avoiding *data races*. Traditionally edge loops follow a *gather-scatter* memory access pattern, that is, edge quantities are computed based on variables at their end nodes, and used to update other locations associated with the node indices. As multiple edges intersect at the nodes, this access pattern will lead to race conditions if left unregulated, next we discuss the methods used to regulate or avoid it.

B. Loop Transformations

In a finite-volume context, edge loops usually express some form of discretized conservation principle (for example computing gradients by application of the divergence theorem, or computing nodal residuals) and as there are no loop-carried dependencies, they can always be converted to point loops that gather information from direct neighbors. The benefits of looping over edges are the reduced computational effort, since edge quantities are computed once, and reduced storage requirements, as to loop over neighbors efficiently an adjacency matrix is needed (which uses more space than the edge graph). However, unless the edge quantities require significant computational effort (FLOP/BYTE $>>$ 1), better performance can be obtained with the neighbor loop version due to the better memory access pattern (fewer loads and stores, and the latter become sequential, and potentially *non-temporal*). Therefore this kind of loop transformation has been adopted for all bandwidth bound routines such as computing gradients and limiters.

C. *Locks* and *Atomic* Operations

The only computationally intensive edge loops in the code are to evaluate convective and viscous fluxes (element loops are discussed later) and here a simple loop transformation would be two times slower. Atomic operations and locks are the typical software-based solutions that can be used to guard the access to potentially colliding memory locations, this has the advantage of requiring few changes to the code. However, for frequently used resources, these solutions have limited scalability, and atomic operations are not defined for the algorithmic differentiation types used in SU2. Therefore, different solutions are needed. Besides domain decomposition, there are two common algorithmic ways to make edge loops thread-safe, coloring and reduction-type strategies.

D. *Reduction*

Our reduction strategy consists in separating the compute and scatter components of edge loops, such that the former does not have to be duplicated. Edge quantities are stored (without scattering) during the normal edge loop, and then reduced for each node independently in a second loop over neighbors, as Fig. 3 illustrates. Alternatively, the target locations (e.g. the residual vector) could be (partially) duplicated, reduction then becomes a simple sum at the expense of a storage overhead proportional to the number of threads.

The reduction approach described above makes the residual loop embarrassingly parallel. However, it has the downside of adding a low compute intensity reduction loop (i.e. low FLOP/BYTE ratio), which in the case of the residual Jacobian matrix involves setting diagonal blocks as the sum of the blocks in the same column, an operation with a very poor memory access pattern as row-major storage is used for this matrix. This reduction operation warrants further explanation, the edge flux (F) contributes to the residual of two nodes (i and j), whose variables are used to computed it, therefore the flux Jacobians ($\partial_i F$ and $\partial_j F$) are used to update four locations of the residual Jacobian matrix ($A_{ii} \leftarrow A_{ii} + \partial_i F$, $A_{ij} \leftarrow \partial_j F$, $A_{jj} \leftarrow A_{jj} - \partial_j F$, $A_{ji} \leftarrow -\partial_i F$). Since the pair (i, j) is unique to each edge, the off-diagonal blocks of the matrix are always safe to write to. It is for the diagonal entries that race conditions may occur, note however that the data required to compute them is naturally stored in the columns, i.e. $A_{ii} = -\sum_{k \neq i} A_{ki}$.

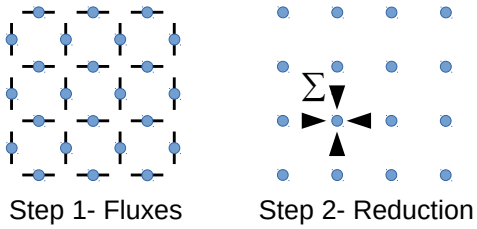


Fig. 3 Reduction strategy to avoid data races in edge loops.

E. Grid *Coloring*

Edge coloring consists in identifying non intersecting groups of edges that can be freely processed in parallel, thereby avoiding the overhead of the reduction loop. Nodes are numbered to reduce the bandwidth of the sparse pattern to improve spatial locality, and edges are sorted by increasing node indices to improve temporal locality when traversing them [4]. Therefore, looping over non intersecting edge groups has a negative impact on temporal locality (increased cache misses), so much so that with simple edge coloring the code performs worse than with the reduction strategy. To preserve some temporal locality we color groups of edges as depicted in Fig. 4. The typical group size is 500 edges and a greedy algorithm is used. This grouping introduces some parallel inefficiency due to the reduction in number of assignable work units within each color. A greater problem with coloring the edges of an unstructured dual grid is doing so with reasonable number of colors, as the maximum number of intersecting edges can easily exceed 30 (a large number of colors also introduces parallel inefficiency by reducing the number of work units). Low parallel efficiency can be especially problematic on coarse multigrid levels due to the reduced mesh size.

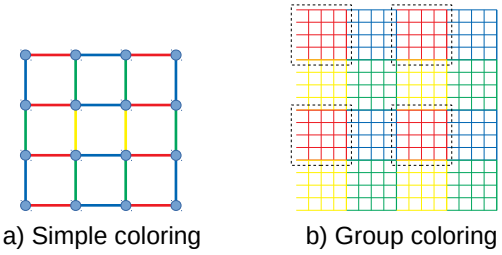


Fig. 4 Simple (single) and edge group coloring.

F. Automatic Fallback Strategies

Group coloring generally gives better performance than the reduction strategy, but in part due to the naive greedy algorithm we implemented, it is less robust and fails on some grids. We attempt to balance performance and user friendliness by automatically switching to the reduction strategy in any MPI rank where the edge coloring results in a parallel efficiency below 0.875 (i.e. one out of eight threads idling due to the quantization of edges). This threshold does not guarantee best performance, different machines have different performance characteristics. This fallback strategy approach requires no code duplication, its implementation consists of a simple choice between scattering (coloring) or storing (reduction) the edge fluxes within the edge loop, followed by a conditional call to the reduction routines outside the edge loop.

G. Finite-Element Applications

In the native structural finite-element (FE) solvers in SU2 [5] (used for nonlinear elasticity simulations and for mesh deformation), element loops are used to populate the stiffness matrix and force vector. Here however, good temporal locality is not as important to performance due to the work-split between discretization and linear solution, i.e. 2 to 3 orders of magnitude more linear iterations are required per nonlinear iteration compared to the flow solvers. Thus we use simple element coloring (which conceptually are small edge groups), which we find to work successfully also for grids with mixed elements. Note that for element loops it is not possible to define a storage-efficient reduction strategy,

instead our fallback strategy consists of using one lock per node to guard the updates of the force vector and stiffness matrix rows.

H. Implementation Details

The interaction between threads and MPI follows a funneled communication model which is widely supported even by older MPI implementations. As depicted in Fig. 5, only the main thread in each rank communicates, however the buffers used for communication are still packed and un-packed by all threads in parallel.

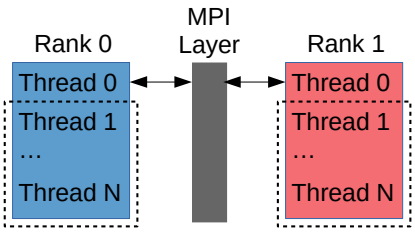


Fig. 5 Funneled communication model, only the main thread interacts with the MPI API.

By default SU2 is compiled without support for hybrid parallelization, to be able to disable the OpenMP directives they are wrapped, this also allows tuning some of them to the version of the standard implemented by the compiler in use. Furthermore, to (cleanly) avoid any overhead, e.g. the indirection introduced by coloring, or the conditionals associated with the fallback strategies, the corresponding booleans are made *constexpr*, and the types that store the coloring information are silently replaced by integer-like types.

I. Numerical Demonstration on Benchmark Problem

To demonstrate the strong scaling capabilities allowed by the hybrid strategy, we consider the benchmark problem from section II.F. Solution times (as previously described) were obtained on a cluster with 24 core nodes (two Intel Xeon E5-2650v4 CPU) connected by an InfiniBand network. Fig. 6 shows the observed speedup (relative to execution on one node) against the expected speedup (simply the number of nodes). On the 24 and 48 core runs, 24 MPI ranks were used, 16 were used on the other three. In the transitions from 24 to 48 cores and 96 to 192 cores the relative speedup is approximately 1.85, which is close to the ideal value of 2. However, from 48 to 96 cores the relative speedup is 1.65, this is due to the automatic switch from coloring to the reduction strategy (which for this problem, on this kind of machine, performs noticeably worse). The final doubling of resources produces almost no speedup, at approximately 2750 nodes (16 500 edges) per core (192 core run) communications dominate and the solution time becomes sensitive even to the position of the cluster nodes in the network topology. The final run with 384 cores also marks the point where one MPI rank spans two NUMA (non-unified memory-access) nodes, which can also negatively impact performance. Finally we note that for this example problem, the solution time would increase in MPI-only mode with more than 48 cores due to reduced effectiveness of the multigrid strategy.

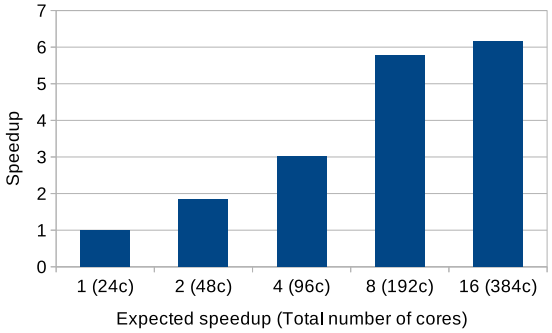


Fig. 6 Observed speedup vs expected speedup with increasing number of compute nodes.

IV. Vectorization

Vectorization is the ability of some hardware to process multiple elements of data with single instructions. Typically with the same latency and throughput, in terms of clock cycles, of their scalar counterparts (i.e. single element). This is known as *single instruction multiple data*, SIMD. Compute-bound routines (FLOP/BYTE >> 1) can expect a speedup proportional to the number of SIMD lanes in the target architecture. In SU2 the most compute intensive task is the computation of convective and viscous fluxes for each edge as described in the previous section. As the number of SIMD lanes available in current hardware (especially GPU's [†] and other "many-core" architectures) matches or exceeds the number of variables per vertex (i.e. the number of conservation equations) it is not worth trying to vectorize the computations for a single edge. Instead, it is more portable and adequate to low-order FVM to simultaneously process one edge per SIMD lane.

Conceptually, and in a C++ implementation context, this can be done by defining a static-size array type for which math functions and operations are *overloaded* such that $\vec{y} = f(\vec{x}) \Rightarrow y_i = f(x_i) \forall\ i$ and similarly for two-argument functions between arrays, and between arrays and scalars. With such an array type available (our implementation consists of 500 lines of code), vectorized code is as natural to write as conventional (scalar) code. Listing 1 shows the computation of a convective flux, function templates are a necessity, as will be explained below, but by and large the template parameters are deduced automatically by the compiler.

```
template<class PrimVarType, class ConsVarType, int nDim>
FORCEINLINE VectorDbl<nDim+2> inviscidProjFlux(const PrimVarType& V,
                                               const ConsVarType& U,
                                               const VectorDbl<nDim>& normal) {
  Double mdot = dot(U.momentum(), normal); // "Double" is the SIMD type
  VectorDbl<nDim+2> flux; // A static-size array of "Doubles"
  flux(0) = mdot;
  for (int iDim = 0; iDim < nDim; ++iDim) {
    flux(iDim+1) = mdot*V.velocity(iDim) + normal(iDim)*V.pressure();
  }
  flux(nDim+1) = mdot*V.enthalpy();
  return flux;
}
```

Listing 1 Snippet of SU2 code illustrating the non-intrusiveness of the vectorization approach.

Exceptions to this simplicity are: 1) gathering the required data for the multiple edges into SIMD format; 2) scattering the results, i.e. fluxes and its Jacobians; 3) and expressing conditional expressions that depend on the solution variables (e.g. *if Mach greater than 1*). These are discussed next in detail.

A. Vectorization Challenges

To vectorize code effectively, conditional expressions need to be made *branch-less* (i.e. expressed without *if statements*). One technique that accomplishes this is *masking*, which consists in evaluating the results of all possible branches and combining them either arithmetically (weighing branches with 0 or 1 and adding) or logically (via bitwise *and* operations with 0x0 or 0xF...F and combination with bitwise *or*). Logical masking is more efficient as the return of SIMD comparison instructions is used directly as the mask, moreover the occurrence of *NaN's* in the inactive branches does not cause issues (0x0 & $NaN = 0$, whereas with arithmetic masking $0 \cdot NaN = NaN$). However, the arithmetic approach is naturally compatible with algorithm differentiation, and so it is the approach used in SU2. We note that the compromise is not significant since most schemes have few conditional expressions.

In an unstructured FVM context, the indirect data accesses via connectivity (edge graph) and adjacency (lists of neighbors) structures also present a challenge. Data that is adjacent in memory can be quickly loaded into SIMD registers, however indirect accesses incur a penalty of typically one order of magnitude due to the higher latency and lower throughput of *gather* instructions and also due to less effective caching. Moreover, the efficiency of *gather* instructions is architecture dependent, even within x86-64 architectures[6]. Therefore, in the interest of portable performance, we avoid those instructions and instead copy portions of data into containers of SIMD types, as shown in Fig. 7. This is akin to transposing a matrix and so it is readily optimized by compilers.

This operation is conveniently encapsulated in the storage class that backs all variable classes, and so the SIMD nature of the data-types only becomes visible in kernel areas of the code. The high-level usage is indistinguishable from scalar code as shown in listing 2.

[†]Note that the execution model of GPU's is formally known as *single instruction multiple thread*, SIMT

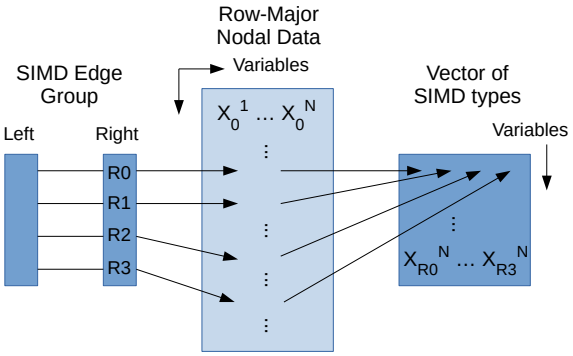


Fig. 7 Graphic representation of the gather operation as implemented in SU2.

```
// Note the data sizes as template parameters, iPoint refers to multiple node indices.
auto myVar = gatherVariables<nVar, nDim>(iPoint, container);
```

Listing 2 High-level statment to retrieve a matrix of SIMD types from a container.

Fluxes and their Jacobians are scattered to the residual vector and system matrix (for implicit solution methods) in similar fashion, again the operations are encapsulated by the respective storage types to hide the low-level details. In summary, code written for the SIMD type will also work with a scalar type.

B. Storage Layout Considerations

For the algorithms in SU2, row-major storage reduces cache misses compared to using one array per variable (i.e. column-major storage) as accessing one of the variables of a node automatically brings some of the others into cache, this makes it the recommended layout for cached machines [7]. However, it makes the vectorization of point loops (e.g. computing primitive variables from the conservative solution variables) more difficult due to the introduction of *strided* accesses. That is, one needs to gather data into temporary column-major variables (instead of a contiguous *load*), perform the computations, and finally scatter the results to the target storage location. On some CPU architectures the cost of these gather/scatter operations can offset the gains from vectorization, especially in bandwidth-bound routines [‡]. Note that data accesses in edge loops always require some form of gather/scatter, regardless of the storage scheme, due to the indirection mentioned before. With row-major storage these operations can be implemented either via *gather* instructions, or via the transposition described above, we leave this choice to the compiler.

So called *array of structures of arrays* storage strategies have been proposed to strike a compromise between locality and ease of vectorization (e.g. in [8]). However, we have found that some well known C++ compilers do not amortize the additional integer arithmetic required to convert n-dimensional coordinates (e.g. (i, j, k) tuples) to the corresponding index in 1D storage space. That is, in a nested loop they often fail to move the relatively expensive modulo arithmetic (associated with the outer dimension) to the outermost loop, instead repeating it in the inner loop. This can of course be tuned by the programmer, by defining iterators to the inner dimensions of the container, but in the interest of keeping the code accessible we deliberately chose to forgo the vectorization of simpler point loops. Furthermore, as mentioned in section II, exploring a different layout would require a total refactoring of the code.

C. An Efficient Implementation

To extract the full potential of the SIMD type, it must be considered that: 1) the number of variables is typically small (< 10), thus it is crucial that data sizes and loop counts be known at compilation to avoid dynamic allocation and perfectly unroll loops, respectively; 2) data accesses are more expensive due to the transposition operation and the higher probability of cache misses, as data from more points needs to be fetched before computations can begin, thus they must be amortized; 3) SIMD registers are scarce compared to general purpose registers, increased register spillage is expected and needs to be mitigated.

Accounting for these three aspects requires dedicated versions of each spatial discretization scheme for 2D and 3D,

[‡]GPU's can be more forgiving of strided accesses, though still greatly benefiting from contiguous (also known as *coalesced*) accesses

with or without viscous fluxes, which results in 4 combinations in total. The 2D/3D versions allow known sizes, whereas the viscid/inviscid versions permit directly re-using (due to *inlining*) some of the data gathered for the convective flux in the viscous routines, and importantly, scattering the results only once. Additionally, we have found that implementing the SIMD type using x86 intrinsic functions and types [9] greatly reduces register spillage. Simple code generation strategies, relying only on the C-preprocessor, are used to generate template specializations of the SIMD type for different SIMD lengths (128, 256, and 512 bits).

Class and function templates are used to create the aforementioned versions of each scheme, we use polymorphism to keep the templates from propagating to the client code of the numerical schemes, i.e. the *solver* classes. That is, all numerical scheme class templates inherit from a conventional abstract base class. This insulates client code from the implementation of the numerical schemes, and delegates the numerical scheme selection logic to a factory method that also serves as the template instantiation point for all 4 combinations mentioned above.

To simplify the interaction with the hybrid parallel implementation, these numerical classes were designed to be thread safe. This means their state variables must be read-only, which precludes passing data to class methods by "member variable", thereby making the implementation more *functional*. That is, code that can be re-used across different families of numerical schemes is implemented in free functions. Likewise, within families of numerical schemes, for which it makes sense to define a class hierarchy, any intermediate results must still be passed explicitly when calling class methods. At the expense of longer function signatures, a new developer can clearly see what are the inputs and outputs of any function or method. Forcing all intermediate computation results to be local, stack-allocated variables, is also a performance consideration as it allows them to be truly temporary, and avoids any *aliasing* considerations associated with heap-allocated variables.

As we wish for all the implementation components of each numerical method to be inline, patterns that rely on conventional polymorphism cannot be used. For example, adding viscous fluxes to a convective method could be done via the *decorator* pattern, whereby functionality is added to an existing object by attaching a decorator object to it. To avoid polymorphism we use a static variant of this pattern, where the decorator class (the viscous fluxes) is set at compilation via a template parameter, and is used as a parent class by the decorated one to allow chaining any number of decorators. Similarly, within a family of schemes, where it would be intuitive to implement large common parts in a parent class and only small details in the final derived classes, we use static polymorphism via the *curiously recurring template pattern*. The pseudo code of listing 3 shows the basic structure of these mechanisms.

```
// The abstract base.
class CNumericsSIMD {
  virtual void ComputeFlux(...) const = 0;
}

// One of the viscous flux decorators.
template<int nDim> class CCompressibleViscousFlux : public CNumericsSIMD {
  void ViscousTerms(...) const {...}
}

// A common parent for centered schemes, with Derived and Base classes as template parameters.
template<class Derived, class Base> class CCenteredBase : public Base {
  void ComputeFlux(...) const final {
    ... // gather data, do some computation
    Derived::DissipationTerms(...); // static polymorphism
    Base::ViscousTerms(...); // static decorator
    ... // scatter results
  }
}

// A final centered scheme, the viscous decorator is a template parameter forwarded to the parent class.
template<class Decorator> class CJSTScheme : public CCenteredBase<CJSTScheme<Decorator>, Decorator> {
  static void DissipationTerms(...) {...}
}

// Template instantiation in factory method, e.g. 3D with viscous fluxes.
CNumericsSIMD* Factory(...) {
  return new CJSTScheme<CCompressibleViscousFlux<3> >(...);
}
```

Listing 3 Pseudo code of the template mechanisms used to efficiently compose the vectorized numerical classes.

Template metaprogramming is perhaps the tallest entry barrier in this implementation, as it is seen by many as a paradigm shift from *declarative programming*. It is however inextricable from modern high performance C++, and there is an ample supply of conference talks [§] covering it in enough depth to get new developers acquainted with its idioms.

D. Interaction with Algorithmic Differentiation

Discrete adjoints (based on algorithmic differentiation, AD) are a central feature of SU2 [10] enabling complex multidisciplinary optimizations [11, 12]. A typical reverse mode AD type contains a floating point primal value, and an integer adjoint index. Therefore, in an array of such types the floating point values are not contiguous and so efficient vectorization is not possible. Nevertheless, when compiling the AD version of SU2 we still use "SIMD arrays" of the AD type as this reduces the memory footprint of discrete adjoint solvers. This is due to the use of *preaccumulation* (where small sections of the AD tape are evaluated during recording, and the resulting Jacobians stored instead of the original statements [13]) and due to the repetition of node indices within SIMD edge groups (that results from ordering the edges) that reduces the size of those Jacobians. For the benchmark problem of the previous section there is a reduction of 29% in the memory used by the AD tape. Therefore, although the AD version is not vectorized, its runtime is still improved as the smaller memory footprint makes the tape faster to evaluate.

E. Numerical Demonstration on Benchmark Problem

Again we use the benchmark problem of previous sections to show the effect of vectorization on the solution time of the code. Table 2 shows the effect of vectorization on the 24 core machine (Broadwell architecture, which supports AVX2 instructions).

Results were also obtained on a machine with support for AVX512, with two Intel Xeon Gold 6248 CPU. The code was compiled with the Intel compiler version 2019.5 and Intel MPI 2019.6, with the optimization flags "-O2 -funroll-loops -qopt-zmm-usage=high -xCOMMON-AVX512", and run in hybrid parallel mode with 10 MPI ranks of 4 threads. Table 3 shows the effect of vectorization on the 40 core machine.

Version	Wall time	Iterations	Time per iteration
7.0.6	237 s	466	0.509 s
7.0.7 (vectorized)	160 s	450	0.354 s

Table 2 Effect of vectorization on the benchmark problem, 24 core machine with AVX2.

Version	Wall time	Iterations	Time per iteration
7.0.6	171 s	490	0.348 s
7.0.7 (vectorized)	97 s	478	0.203 s

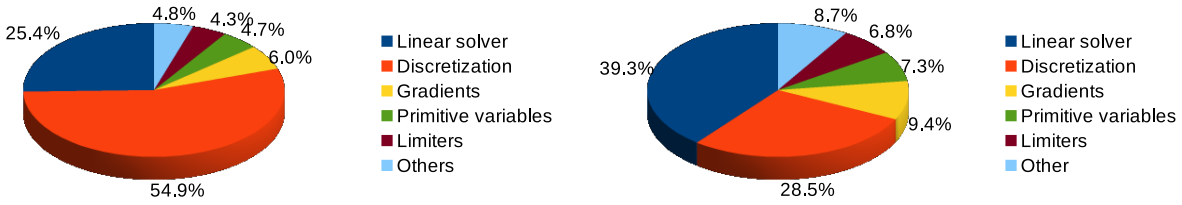
Table 3 Effect of vectorization on the benchmark problem, 40 core machine with AVX512.

On the 24 core machine the speedup is 1.44, whereas on the 40 core machine the speedup is larger at 1.71, which is expected due to the increased SIMD length (8 vs 4 doubles) and number of AVX registers (32 vs 16) which reduces spillage. Note how the use of vectorization on the older architecture is equivalent to a generational improvement in hardware.

It is worth noting that the dedicated 2D/3D, viscid/inviscid versions of the schemes also contribute to improved performance, and that speedups of 4 and 8 (for AVX2 and AVX512 respectively) were not expected since only part of the code was vectorized (moreover it is known that low order FVM is ultimately bandwidth-bound). Finally, we investigate how the computational time is divided across the main tasks of the flow solver (i.e. excluding turbulence which accounts for less than 20% of the total). This was done by profiling the code running on similar compute-to-memory conditions as on the 24 core machine but without MPI. Figures 8a and 8b show the scalar and vectorized results, respectively. Vectorization takes the discretization task (computing fluxes) from using over 55% of the time to only 28.5%, making

[§] For example https://cppcon.org/

the linear solver the most costly operation in the vectorized implementation even in a problem that requires only 3 linear iterations.



(a) Scalar implementation. **(b) Vectorized (AVX) implementation.**

Fig. 8 Percentage time spent in different tasks of the flow solver.

V. Algorithmic Improvements

Finally we overview a number of recent upgrades to the numerical algorithms in SU2, available from version 7.0.0, that can result in up to an order of magnitude improvement in time-to-solution, while greatly enhancing the robustness and scalability of the flow solver. With these we seek to increase the convergence rate and eliminate catastrophic failures of the solver to ensure a robust path to nonlinear convergence of the governing equations, which is desirable for both steady state problems as well as the inner loops of unsteady problems treated with the dual time-stepping approach. For example, difficult transients at solution startup (e.g., steep gradients at no-slip walls) can cause jumps in the solution variables that result in non-physical states, which can crash the solver or severely limit the allowable CFL number. In these cases, users often rely on CFL ramping strategies and slope limiters to keep the solution stable (at a cost of degraded convergence), or other measures, such as activating second-order convective schemes only after converging the problem to some degree with first-order schemes. It can be difficult to tune these processes manually, and instead, we have implemented a more sophisticated controller for nonlinear iterations that is completely automated.

A. Realizability

While second-order convective schemes are needed to obtain the accuracy required by most modern applications, they are a primary source of instability or convergence stall in solvers that rely on a MUSCL-type approach for upwind schemes. In the MUSCL approach for vertex-based schemes [14], the flow state is reconstructed using the gradients of the variables at each node to project a higher-order approximation of the state at the mid-point of an edge (dual cell interface) for use in the flux calculation. For second-order accuracy, a linear approximation is applied:

$$V_L \approx V_i + \phi \left(\nabla V|_i \cdot \bar{r}_{ij} \right), \tag{1}$$

$$V_R \approx V_j + \phi \left(\nabla V|_j \cdot \bar{r}_{ji} \right), \tag{2}$$

where V is the vector of primitive variables, $\bar{r}_{ij}$ is the vector pointing from node i to the midpoint of edge ij, $\bar{r}_{ji}$ is the vector pointing from node j to the midpoint of edge ij, and we have also introduced a slope limiter that is represented by ϕ to ensure a Total Variation Diminishing (TVD) scheme. The nodal gradients are typically computed with either a Green-Gauss (GG) or least-squares method [15].

Following [16], we implement physical realizability checks on V_L and V_R after reconstruction of the primitive variables to the interface. If the density, pressure, or sound speed is found to be negative after reconstruction, then the values at the vertices are used directly for the flux calculation. This results in a locally first-order accurate approximation that will avoid a catastrophic failure of the solver in the ensuing flux calculation. The state on each side of the interface is checked independently, and if a non-realizable state is found, its owning vertex is flagged as first-order in the following iterations until achieving 20 consecutive physically realizable reconstructions, at which point the vertex is reactivated as second-order. In practice, only a small number of vertices report as first-order temporarily, typically during solution startup.

We have also implemented the capability to perform the reconstruction with a separate gradient method from that used for computing the gradients needed for viscous fluxes and source terms. For example, it is possible to perform reconstruction with an unweighted least-squares (LSQ) method while applying GG or an inverse-distance weighted

least-squares (WLSQ) method for the viscous and source terms. Application of an unweighted least-squares method for reconstruction can improve robustness for some problems. However, while there is evidence that LSQ is superior for reconstructing nonlinear data at interfaces [17], it has also been shown to be inferior to GG or WLSQ for evaluating the actual gradient values [18] and should be avoided outside of reconstruction.

In addition to checking realizability of reconstructed states, we also check for non-physical values of the primitive variables after each nonlinear solution update before starting a new iteration. If density / pressure / speed of sound / temperature are found to be negative at a vertex, we simply cancel the solution update and reuse the state from the previous iteration.

B. Under-relaxation

We apply implicit time integration schemes for efficient convergence to steady state. After discretizing the governing equations in space, one obtains the coupled system of ordinary differential equations written as

$$\frac{d}{d\tau}\left(U_i|\Omega_i|\right) + \mathcal{R}_i(U) = 0, \tag{3}$$

where $\mathcal{R}_i(U)$ is the numerical residual (a function of the conservative variables U) that represents the integrated sum of all spatial terms for the control volume surrounding vertex i. $|\Omega_i|$ is assumed constant in time for now. As we are pursuing steady state solutions, the time variable is interpreted as a pseudo time variable τ.

Let time level n correspond 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 $\Delta\tau$ where $\Delta\tau = \tau^{n+1} - \tau^n$. We also define $\Delta U_i = U_i^{n+1} - U_i^n$. For an implicit scheme, such as backward Euler, the residual is evaluated using the solution state at the new time level U^{n+1}. Applying this to Eqn. (3), one has

$$\Gamma_i \frac{\Delta U_i}{\Delta\tau_i}|\Omega_i| = -\mathcal{R}_i(U^{n+1}). \tag{4}$$

A first-order linearization about time level n provides an approximation to $\mathcal{R}_i(U^{n+1})$:

$$\mathcal{R}_i(U^{n+1}) = \mathcal{R}_i(U^n) + \sum_{j \in \mathcal{N}(i) \cup i} \frac{\partial \mathcal{R}_i(U^n)}{\partial U_j}\Delta U_j^n + \mathcal{O}(\Delta\tau^2), \tag{5}$$

where $\mathcal{N}(i)$ here also includes point i, i.e., Jacobian contributions for the impact of changes at i on itself. Introducing Eqn. (5) into Eqn. (4), we find that the following linear system should be solved to obtain the solution update (ΔU_i^n):

$$\left(D\delta_{ij} + \frac{\partial \mathcal{R}_i(U^n)}{\partial U_j}\right)\Delta U_j^n = -\mathcal{R}_i(U^n), \tag{6}$$

where repeated j implies summation over $j \in \mathcal{N}(i) \cup i$, $D = \Gamma_i \frac{|\Omega_i|}{\Delta\tau_i^n}$, and $\delta_{ij} = 1$ if $i = j$ and is zero otherwise, i.e., D is a block contribution to the diagonal of the system for each point i. In practice, the Jacobian terms $\frac{\partial \mathcal{R}}{\partial U}$ in Eqn. (6) contain approximations that reduce the computational effort or memory footprint required for constructing the linear system, e.g., only first neighbors are considered in the stencil.

The solution is relaxed to a steady state iteratively by approximately solving Eqn. (6) with a suitable time step $\Delta\tau_i^n$ to obtain a solution update ΔU^n. To improve the robustness of the implicit solution process, as is commonly done, we introduce an under-relaxation parameter ω with bounds $[0, 1]$ that is applied to our nonlinear solution updates resulting from the approximate solution of Eqn. (6) as

$$U_i^{n+1} = U_i^n + \omega^n \Delta U_i, \tag{7}$$

where ω is selected automatically by restricting the change in the density and total energy variables to a fixed percentage of the value in the previous iteration. Several authors have applied a similar strategy [19, 20] and report bounding the updates to 10% or 20% of the previous iteration value. In our implementation, we choose 20% as the default, and we define ω as a local parameter applied to the solution update at each vertex, rather than a fixed constant in space. For RANS problems with the Spalart-Allmaras model, we apply the same under-relaxation approach to the turbulence variable updates with a larger bound of 99%. The minimum ω among those for the density, total energy, or turbulence equations is chosen for each vertex. If ω goes below a very small tolerance (e.g., 1e-10), then ω is set to 0.0 to cancel the solution update entirely at that vertex.

C. CFL Adaption

Implicit schemes offer improved stability conditions, and thus allow for higher CFL numbers and faster convergence. However, manually tuning the CFL or defining CFL ramping schedules that maximize performance can be difficult or time-consuming. The introduction of the under-relaxation parameter offers a useful metric for defining an automatic CFL adaption strategy: we apply the exponential progression with under-relaxation technique for evolving the CFL number during the calculation.

In this approach, updates to the CFL number are coupled to the under-relaxation parameter with each nonlinear solution update, e.g., if a small ω is required to limit the solution update, then the CFL is automatically reduced for the next nonlinear iteration. A reliable CFL update strategy is given by [20, 21]:

$$\mathrm{CFL}^{n+1} = \begin{cases} \alpha \cdot \mathrm{CFL}^n, & \omega^n = 1.0 \\ \mathrm{CFL}^n, & \omega_{min} < \omega^n < 1.0 \\ \beta \cdot \mathrm{CFL}^n, & \omega^n < \omega_{min} \end{cases} \tag{8}$$

where we choose $\omega_{min} = 0.1$, $\alpha > 1.0$, and $\beta < 1.0$. The CFL values are also bounded by user-specified min and max values. Recall that ω is defined locally in each control volume, and therefore, we also define a local CFL number in each control volume, which improves convergence in the case where only a small number of cells could stall the entire solution due to global under-relaxation and CFL values [22].

Based on numerical experiments, $\beta = 0.1$ and $\alpha = 2.0$ perform well for inviscid and laminar Navier-Stokes problems. For RANS problems, which are solved in a segregated manner, a more conservative increase with $\alpha = 1.2$ have proved beneficial. Fig. 9 shows residual convergence for a wide selection of typical test cases that are provided with SU2 [¶]. The combination of all enhancements described above for nonlinear convergence enables the use of essentially infinite CFL numbers for inviscid and laminar problems (10^6 or greater) and CFLs up to approximately 1000 for RANS problems, resulting in time-to-solution speedups of an order of magnitude or more for many cases relative to previous versions of the software.

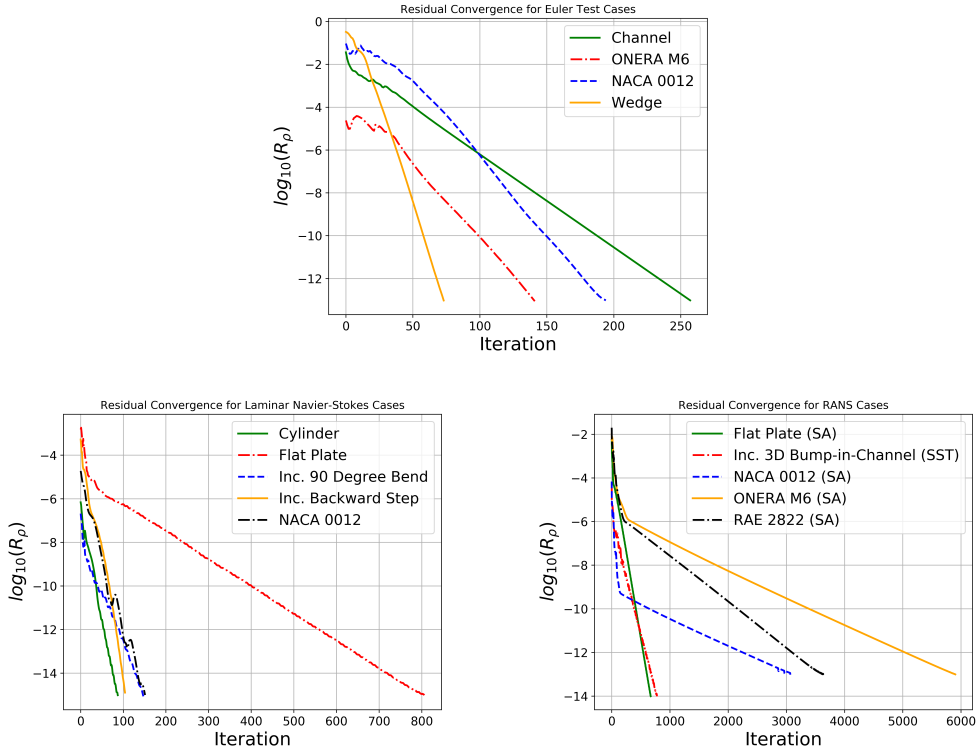


Fig. 9 Results of applying the improved nonlinear iteration controller for typical test cases.

[¶]https://github.com/su2code

VI. Numerical Examples

A. NACA 0012 Airfoil

We focus first, on NASA's NACA0012 verification case (Mach 0.15, 300 K, 10 deg AoA, Re $6 \cdot 10^6$) [23] to ensure the performance optimizations did not impair the accuracy of the code. This problem was chosen since grid-converged solutions from a number of well known CFD codes lie on a narrow band. The 6 finest "Family II" reference grids [24] for this geometry were used. Convection was discretized with Roe's scheme without low Mach number corrections, using a second order MUSCL reconstruction based on gradients computed by the Green-Gauss method, and limited using Van-Albada's limiter. The SA turbulence model was used in the "noft2" variant, with first order upwind discretization of its convective terms. Table 4 shows the lift and drag coefficients obtained. Fitting 3-parameter equations of the form $y(h) = y_0 + Ch^n$, with h a representative grid spacing, to the first 5 data points suggests fully resolved values (y_0) of 1.0911 and 0.012247 for the lift and drag coefficients respectively. Both values agree well with published values [25].

Grid level	C_l [-]	C_d [-]
1	1.09117	0.0122547
2	1.09097	0.0122500
3	1.09052	0.0122648
4	1.09004	0.0124046
5	1.08778	0.0132067
6	1.07477	0.0178016

Table 4 Lift and drag coefficients for the 6 finest "Family II" NACA0012 reference grids.

B. NASA's High Lift Common Research Model

The main purpose of the performance and robustness improvements presented in this paper is to enable large scale simulations and design optimization in SU2. As an illustrative example of that, we choose the high-lift configuration of NASA's Common Research Model (HLCRM), shown in Fig. 10, which has been extensively described and studied in the literature, e.g. [26]. For this problem, alongside the simulation results, we provide solution times for reference. Furthermore, we derive best practice guidelines for how the hybrid parallel implementation should be used.

The "B3-HLCRM" reference grids were used, these are four mixed element grids (from tetrahedra to hexahedra) with 8, 27, 71, and 208 million nodes. The SA turbulence model was used ("noft2" variant) with first order upwind convection, and the flow equations discretized with the JST scheme with 0.5 and 0.01 second and fourth order dissipation coefficients respectively. Table 5 shows the lift and drag coefficients on the four grids, which agree well with published numerical results [26]. On 10 of the 40-core AVX512 nodes previously mentioned, the fine grid converges from free-stream conditions after 23 000 iterations in 28.5 h. After 15 000 iterations and 5 orders of magnitude reduction of the continuity residual, the coefficients settle to within 4 significant digits. The behavior is similar on the other grid levels, with coarser levels requiring fewer iterations, e.g. 14 000 for convergence on the coarse grid, and vice-versa. The number of iterations is comparable to what has been published for other codes on similar problems [27, 28], when using similar solution methods[‖]. Notwithstanding, the multigrid agglomeration strategy in SU2 must be further improved to cope with the highly stretched grids that are characteristic of these problems.

To demonstrate the trade-offs between distributed and shared memory parallelization approaches, the average time per iteration was measured for the coarse grid with varying number of OpenMP threads per MPI rank on 192 and 408 total cores (8 and 17 of the aforementioned 24 core nodes respectively). The results are shown in Fig. 11 With few threads per rank the domain decomposition overhead is highest (static load imbalance, communication costs, etc.), whereas with few MPI ranks per compute node, i.e. many threads, the shared memory overhead increases (coloring/reduction, cache invalidation by *false sharing*, etc.).

For this system and problem, the solution time is minimized with one MPI rank per socket (also NUMA nodes in this system), we expect this to be so anytime a significant number of compute nodes is used, especially with lower performance networks. Recent x86 CPU architectures have up to 64 cores per NUMA node, here 2 to 4 MPI ranks could

[‖] We are not aware of solution times from other solvers.

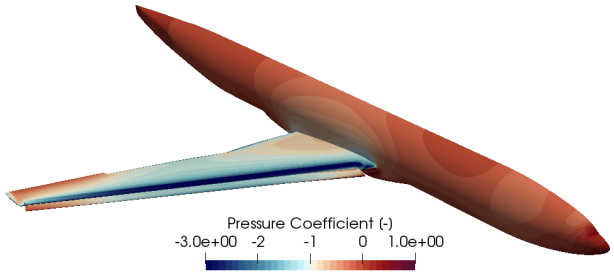


Fig. 10 Pressure coefficient contours on the HLCRM at 8 deg AoA, medium grid.

B3-HLCRM Grid	C_L [-]	C_D [-]
Extra Fine	1.74265	0.169493
Fine	1.73726	0.169452
Medium	1.73315	0.169939
Coarse	1.72186	0.170174

Table 5 Lift and drag coefficients of the HLCRM at 8 deg AoA.

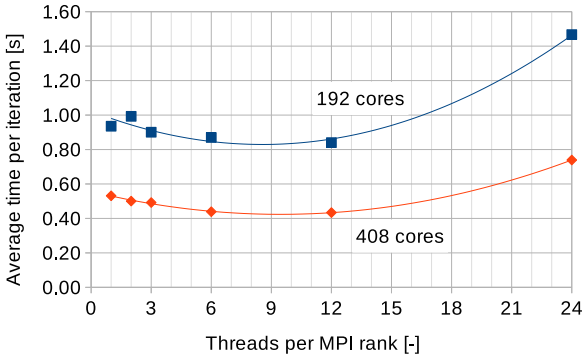


Fig. 11 Average time per iteration against number of threads per rank (2-socket 24 core nodes).

be optimal especially if it avoids falling back to the reduction strategy. As observed with the benchmark problem, there is a penalty for covering 2 NUMA nodes with 1 MPI rank (24 threads in this case) therefore, the number of ranks should always be a multiple of the number of NUMA nodes. From 192 to 408 cores at 12 threads per rank the parallel efficiency is 91.2%. At those core counts, the hybrid parallelization allows a speedup of 1.11 and 1.22 respectively, compared to plain MPI. This correlation between number of nodes and speedup is expected, and justified by the heterogeneity of communication times within nodes or over the network. However, within a single machine, even if multi-socket, we expect plain MPI to perform better insofar as the algorithms used remain effective with the number of partitions used. With 408 cores the solution time improves by using 2 threads rather than 1, whereas with 192 cores it worsens initially and only improves with 3 threads. A number of factors explains this: a) with 192 cores the communication cost is lower as all nodes are on the same network level, i.e. connected to the same switch; b) load imbalance issues are less important as MPI domains are still relatively large; c) finally, for this problem the cost of switching to the hybrid approach is significant as the reduction strategy must immediately (2 threads) be used by the majority of MPI domains.

VII. Concluding Remarks

We have presented a number of software implementation optimizations responsible for an average 4-fold performance increase of SU2. These optimizations all build upon each other, for example, it would not have been so advantageous to vectorize the flux computations while keeping the linear system in double precision. Similarly, without the data layout refactoring, the preprocessing-type operations (computing gradients, limiters, and primitive variables) would be two times slower and it would be nearly impossible to implement vectorization cleanly and efficiently. A high performance but unstable solver is not a desired outcome, and so effort has been put into improving the robustness of SU2 by means of: 1) reconstruction realizability checks; 2) automatic under-relaxation; 3) and automatic CFL adaptation. For some problems this results in 10-fold better performance by dispensing with substantial manual initialization and ramping strategies. Finally we presented results on benchmark problems known to the aeronautics community, some of which were not possible to solve with SU2 prior to this work. The numerical results, and number of iterations to solution (the proxy we have for solution time) are inline with published results from well known CFD codes.

Acknowledgements

We are grateful to the UK Materials and Molecular Modelling Hub, and to the UK Turbulence Consortium, for computational resources, which are partially funded by EPSRC (EP/T022213/1 and EP/R029326/1, respectively). The first author thanks EPSRC (EP/R007470/1) for partly funding this research.

References

[1] Economon, T. D., Palacios, F., Copeland, S. R., Lukaczyk, T. W., and Alonso, J. J., "SU2: An Open-Source Suite for Multiphysics Simulation and Design," *AIAA Journal*, Vol. 54, No. 3, 2016, pp. 828–846. doi:10.2514/1.J053813.

[2] Karypis, G., Schloegel, K., and Kumar, V., "Parmetis: Parallel graph partitioning and sparse matrix ordering library," *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[3] Stepanov, A. A., and Rose, D. E., *From Mathematics to Generic Programming*, 1st ed., Addison-Wesley Professional, 2014.

[4] Economon, T. D., Mudigere, D., Bansal, G., Heinecke, A., Palacios, F., Park, J., Smelyanskiy, M., Alonso, J. J., and Dubey, P., "Performance optimizations for scalable implicit RANS calculations with SU2," *Computers & Fluids*, Vol. 129, 2016, pp. 146 – 158. doi:https://doi.org/10.1016/j.compfluid.2016.02.003.

[5] Sanchez, R., Albring, T., Palacios, R., Gauger, N. R., Economon, T. D., and Alonso, J. J., "Coupled adjoint-based sensitivities in large-displacement fluid-structure interaction using algorithmic differentiation," *International Journal for Numerical Methods in Engineering*, Vol. 113, No. 7, 2018, pp. 1081–1107. doi:10.1002/nme.5700.

[6] Fog, A., *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*, Technical University of Denmark, 2020.

[7] Lohner, R., *Efficient Use of Computer Hardware*, John Wiley & Sons, Ltd, 2008. doi:10.1002/9780470989746.

[8] Hadade, I., Wang, F., Carnevale, M., and di Mare, L., "Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures," *Computer Physics Communications*, Vol. 235, 2019, pp. 305–323. doi:10.1016/j.cpc.2018.07.001.

[9] "Intrinsics Guide," , Jun 2020. URL `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[10] Albring, T. A., Sagebaum, M., and Gauger, N. R., "Efficient Aerodynamic Design using the Discrete Adjoint Method in SU2," *17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2016, pp. 13–17. doi:10.2514/6.2016-3518.

[11] Burghardt, O., Gauger, N. R., Gomes, P., Palacios, R., Kattmann, T., and Economon, T. D., "Coupled Discrete Adjoints for Multiphysics in SU2," *AIAA AVIATION 2020 FORUM*, 2020. doi:10.2514/6.2020-3139.

[12] Gomes, P., and Palacios, R., "Aerodynamic-driven topology optimization of compliant airfoils," *Structural and Multidisciplinary Optimization*, Vol. 62, 2020, pp. 2117–2130. doi:10.1007/s00158-020-02600-9.

[13] Sagebaum, M., Albring, T., and Gauger, N. R., "High-performance derivative computations using CoDiPack," *ACM Transactions on Mathematical Software*, Vol. 45, No. 4, 2019. doi:10.1145/3356900.

[14] van Leer, B., "Towards the Ultimate Conservative Difference Scheme. V. A Second-order Sequel to Godunov's Method," *Journal of Computational Physics*, Vol. 32, No. 1, 1979, pp. 101–136. doi:10.1016/0021-9991(79)90145-1.

[15] Blazek, J., *Computational Fluid Dynamics: Principles and Applications*, Elsevier, Oxford, 2005.

[16] Pandya, M. J., Diskin, B., Thomas, J. L., and Frink, N. T., "Improved Convergence and Robustness of USM3D Solutions on Mixed-Element Grids," *AIAA Journal*, Vol. 54, No. 9, 2016, pp. 2589–2610. doi:10.2514/1.J054545.

[17] Anderson, W., and Bonhaus, D. L., "An implicit upwind algorithm for computing turbulent flows on unstructured grids," *Computers & Fluids*, Vol. 23, No. 1, 1994, pp. 1 – 21. doi:https://doi.org/10.1016/0045-7930(94)90023-X.

[18] Mavriplis, D. J., "Revisiting the Least-Squares Procedure for Gradient Reconstruction on Unstructured Meshes," *AIAA Paper 2003-3986*, 2003. doi:doi:10.2514/6.2003-3986.

[19] Nishikawa, H., Diskin, B., Thomas, J., and Hammond, D., "Recent Advances in Agglomerated Multigrid," *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, 2013. doi:10.2514/6.2013-863.

[20] Burgess, N., "Advances in Numerical Methods for CREATE-AV Analysis Tools," *52nd Aerospace Sciences Meeting*, 2014. doi:10.2514/6.2014-0417.

[21] Ceze, M., and Fidkowski, K., "A Robust Adaptive Solution Strategy for High-Order Implicit CFD Solvers," *20th AIAA Computational Fluid Dynamics Conference*, 2011. doi:10.2514/6.2011-3696.

[22] Menier, V., "Numerical methods and mesh adaptation for reliable rans simulations," Ph.D. thesis, Mathematical Physics, Université Pierre et Marie Curie - Paris VI, 2015.

[23] Rumsey, C., "2D NACA 0012 Airfoil Validation," , Aug 2020. URL `https://turbmodels.larc.nasa.gov/naca0012_val.html`.

[24] Rumsey, C., "Grids - NACA 0012 Airfoil for Turbulence Model Numerical Analysis," , Nov 2019. URL `https://turbmodels.larc.nasa.gov/naca0012numerics_grids.html`.

[25] Diskin, B., Thomas, J. L., Rumsey, C. L., and Schwöppe, A., "Grid-Convergence of Reynolds-Averaged Navier–Stokes Solutions for Benchmark Flows in Two Dimensions," *AIAA Journal*, Vol. 54, No. 9, 2016, pp. 2563–2588. doi:10.2514/1.J054555.

[26] Rumsey, C. L., Slotnick, J. P., and Sclafani, A. J., "Overview and Summary of the Third AIAA High Lift Prediction Workshop," *Journal of Aircraft*, Vol. 56, No. 2, 2019, pp. 621–644. doi:10.2514/1.C034940.

[27] Kroll, N., Langer, S., and Schwöppe, A., "The DLR Flow Solver TAU - Status and Recent Algorithmic Developments," *52nd Aerospace Sciences Meeting*, 2014. doi:10.2514/6.2014-0080.

[28] Rivers, M. S., Hunter, C., and Vatsa, V. N., "Computational Fluid Dynamics Analyses for the High-Lift Common Research Model Using the USM3D and FUN3D Flow Solvers," *55th AIAA Aerospace Sciences Meeting*, 2017. doi:10.2514/6.2017-0320.