Exploring Shared-memory Optimizations for an Unstructured Mesh CFD Application on Modern Parallel Systems

Dheevatsa Mudigere*, Srinivas Sridharan*, Anand Deshpande*,
Jongsoo Park†, Alexander Heinecke†, Mikhail Smelyanskiy†, Bharat Kaul*, Pradeep Dubey†,
Dinesh Kaushik‡, David Keyes§

*Parallel Computing Lab, Intel Corporation, Bangalore, India
†Parallel Computing Lab, Intel Corporation, Santa Clara, CA
‡Qatar Foundation, Doha, Qatar
§King Abdullah University of Science and Technology, Thuwal, Saudi Arabia

***Abstract*—In this work, we revisit the 1999 Gordon Bell Prize winning PETSc-FUN3D aerodynamics code, extending it with highly-tuned shared-memory parallelization and detailed performance analysis on modern highly parallel architectures. An unstructured-grid implicit flow solver, which forms the backbone of computational aerodynamics, poses particular challenges due to its large irregular working sets, unstructured memory accesses, and variable/limited amount of parallelism. This code, based on a domain decomposition approach, exposes tradeoffs between the number of threads assigned to each MPI-rank subdomain, and the total number of domains. By applying several algorithm- and architecture-aware optimization techniques for unstructured grids, we show a 6.9X speed-up in performance on a single-node Intel® Xeon™[1] E5 2690v2 processor relative to the out-of-the-box compilation. Our scaling studies on TACC Stampede supercomputer show that our optimizations continue to provide performance benefits over baseline implementation as we scale up to 256 nodes.**

***Keywords*—CFD, Krylov Solver, Multi-core, OpenMP+MPI**

I. Introduction

Computational aerodynamics is both an important and a representative workload for high performance computing, frequently employed in benchmarking and tuning hardware and software-programming environments. Scaling an unstructured-grid implicit flow solver to a large number of distributed nodes has been a staple of HPC since the ascent of commercial parallel processing in the mid-1980's. The performance of these codes have been pushed beyond tens of thousands of processors through scalable iterative methods, such as Newton-Krylov with domain-decomposed multilevel preconditioners, which sustain a fixed ratio of neighbor communication to bulk processing while the impact of global synchronization grows slowly under careful load balancing. However, the fundamental design challenges faced by conventional uni-processor architectures over the past decade compelled the industry to undergo a tectonic shift in the direction of multi- and many-core architectures. Practitioners are now forced to look at scaling their workloads not only across distributed memories, but also across large numbers of cores of modern massively threaded single shared-memory nodes.

The earliest single node studies have naturally focused on structured grids. However, an unstructured-grid solver comprises of a diverse range of compute- and memory-bound kernels, with variable degree of parallelism, which makes achieving high single-node parallel efficiency a very challenging task. Single-node optimization for unstructured grid applications is relatively nascent and actively being studied. Duffy *et al.* [1], have evaluated leveraging fine-grained parallelism with early GPUs for the NASA FUN3D code, specifically accelerating only the point implicit solver of FUN3D on GPUs.

In this work, we extend the investigation of computational aerodynamics community into strong shared-memory scaling by using the PETSc version of FUN3D (referred to as PETSc-FUN3D in this paper) as a case study. PETSc-FUN3D is chosen since its performance was modeled and measured in a 1999 Gordon Bell Prize paper [2], [3] and revisited in the early days of many-core processing [4]. We study an incompressible flow solver over an aircraft wing geometry, meshed using an unstructured grid as our data set. Unstructured-grid codes differ from structured grid codes profoundly in irregular data accesses, a harder to capture working set, as well as variable and limited amount of instruction-, vector- and thread-level parallelism.

This paper makes the following contributions:

- We demonstrate a number of shared memory optimization techniques applied to key computational kernels, and show performance benefits both at kernel and application level. We achieve a $6.9X$ increase in performance for the full application on a single-node Intel Xeon E5 2690 processor, relative to the base version.

[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

- Further, we show that our optimizations continue to provide performance benefits as we scale the application to multiple nodes and establish the scaling limits of the Netwon-Krylov-Schwarz solver on today's highly parallel architectures.
- We evaluate and compare the explicit (MPI-only) parallelization and hybrid (MPI+OpenMP) parallelization approaches, and summarize the findings towards extracting both coarse- and fine- grained parallelism for a complex and large-scale (CFD) application such as PETSc-FUN3D.
- We carry our performance optimization work within the PETSc [5]–[7] framework, a widely used solver library, thus making benefits of our work extensible to the larger CFD community. We propose several improvements to the PETSC communication routines to further improve OpenMP scalability.

This paper is organized as follows. We begin with background information on the discretization scheme, governing equations and preconditioned Newton-Krylov implicit solver in Section 2. In Section 3, we describe the main computational kernels and their optimization challenges. In Section 4, we describe the experimental setup and the dataset used, as well as the performance profile of the baseline code. We describe various key optimizations and performance results for the main computational kernels in Section 5. Section 6 contains single-node as well as multi-node optimization results for the full PETSc-FUN3D application. We conclude in Section 7 along with our plan for future work.

II. Background

A. Control-volume Discretization for Euler Flow

FUN3D is a tetrahedral, vertex-centered unstructured mesh code for solving the compressible and incompressible Euler and Navier-Stokes equations originally developed by W.K. Anderson of the NASA Langley Research Center [8], [9]. FUN3D has been used for design optimization of airplanes, automobiles, and submarines with irregular meshes comprising several million mesh points. Our work is based on the PETSc version of FUN3D, namely the PETSc-FUN3D, although hereafter in this article we refer to it as just FUN3D. Our interest in this article is limited to the performance optimization of the forward solver in the inviscid incompressible regime of an ideal gas. This regime poses the greatest challenge for high performance, since it reduces the physics to the minimum complexity. When compressibility, viscosity, and real gas effects are activated, the computation becomes more flop-intensive without significantly expanding the memory traffic or communication, and without any fundamental change in the solution algorithm beyond parameter tuning.

1) Discretization: In space, FUN3D uses a control volume discretization with a variable-order flux-difference Roe scheme [10] for approximating the convective fluxes and a Galerkin discretization for the viscous terms. The control volumes are centered on the vertices and their boundaries ∂V are formed by a dual mesh with faces that bisect the edges between the vertices, such that each volume element contains all of the points closest to the vertex at its center. For incompressible flow in three dimensions, computing the limited numerical fluxes requires solving a 3×3 eigen-system on each face of the volume element. (For compressible flows in three dimensions, this eigen-system becomes 5×5.) This discretization which is widely used in CFD codes [8] is relatively insensitive to mesh stretching and is second-order accurate in space, in regions of smooth meshes and solutions. For time discretization, the classic CFD pseudo-transient time-stepping due to Mulder and Van Leer [11] is used.

2) Governing Equations: The incompressible governing equations employed in FUN3D are written in integral form [8] as

$$V\frac{\partial \mathbf{q}}{\partial t} + \int_{\partial V} \mathbf{f}_i \cdot \hat{n} dA - \int_{\partial V} \mathbf{f}_v \cdot \hat{n} dA = 0 \tag{1}$$

where $\mathbf{q}$ is the vector of state variables, $\mathbf{q} = (p, u, v, w)^T$, the inviscid flux is $\mathbf{f}_i \cdot \hat{n} = (\beta\Theta, u\Theta + n_x p, v\Theta + n_y p, w\Theta + n_z p)^T$ and the viscous flux is $\mathbf{f}_v \cdot \hat{n} = (0, n_x\tau_{xx} + n_y\tau_{xy} + n_z\tau_{xz}, n_x\tau_{xy} + n_y\tau_{yy} + n_z\tau_{yz}, n_x\tau_{xz} + n_y\tau_{yz} + n_z\tau_z z)^T$,

where, in turn, (u, v, w) are the velocity components in the three Cartesian directions, p is the pressure, β is an artificial compressibility parameter, and $\Theta \equiv n_x u + n_y v + n_z w$ is the velocity normal to the surface of the control volume. The Euler setting omits the viscous fluxes.

B. Parallel Implicit Nonlinear Solver: Newton-Krylov-Schwarz (NKS)

Implicit solvers are required for problems that are transient but temporally stiff or steady-state. In the former case, we set up an implicit problem on each time step, which can be much larger than the largest time step that satisfies the Courant-Friedrichs-Lewy stability restriction. In the latter case, for nonlinear problems, we also set up a series of implicit problems on time steps chosen artificially as nonlinear continuation parameters, in order to attempt to "globalize" the convergence of Newton's method or obtain convergence from an arbitrary initial condition. As the time step approaches infinity, the steady-state solution is achieved. In either case, the typical implicit step has the form:

$$\mathbf{F}(\mathbf{u}^l) \equiv \frac{1}{\Delta t^l}(\mathbf{u}^l - \mathbf{u}^{l-1}) + \mathbf{f}(\mathbf{u}^l) = 0 \tag{2}$$

where $\Delta t^l \to \infty$ as $l \to \infty$, $\mathbf{u}^l$ represents the fully coupled vector of unknowns, and $\mathbf{f}(\mathbf{u}^l)$ is the vector of nonlinear conservation laws arising from the spatial discretization of the governing equations. An initial guess, $\mathbf{u}^0$, is supplied and each member of the sequence of nonlinear problems, $l = 1, 2, \ldots$, is solved with an inexact Newton method for $\mathbf{u}^l$ using inner iteration k inside of the time step, e.g.:

$$\| \mathbf{F}(\mathbf{u}^{l,k-1}) + \mathbf{F}'(\mathbf{u}^{l,k-1})\delta\mathbf{u}^{l,k} \| < \epsilon \tag{3}$$

followed by $\mathbf{u}^{l,k} = \mathbf{u}^{l,k-1} + \delta\mathbf{u}^{l,k}, k = 1, 2, \ldots$, until satisfied for sufficiently small ϵ. In turn, the linear systems $A\mathbf{x} = \mathbf{b}$, where $\mathbf{x} = \delta\mathbf{u}^{l,k}, \mathbf{b} = -\mathbf{F}(\mathbf{u}^{l,k-1})$, and $A = \mathbf{F}'(\mathbf{u}^{l,k-1})$, for the Newton corrections are solved with a Krylov method. In our case, we rely directly on matrix-free Jacobian-vector product operations to approximate the action of the Jacobian matrix on Krylov vectors, as described in [12]. The Krylov method needs to be preconditioned for acceptable inner iteration convergence rates, and the preconditioning can be the "make-or-break" feature of an implicit code. A good preconditioner saves time and space by permitting fewer iterations in the Krylov loop and smaller storage for the Krylov subspace. An additive Schwarz preconditioner can accomplish this in a concurrent, subdomain-by-subdomain manner, with an approximate solve in each subdomain of a matrix constructed as an incomplete factorization of the Jacobian of the local domain. Theory and application of Schwarz domain decomposition methods for PDEs can be found in [13]. The coefficients for the preconditioning operator are derived from a lower-order, sparser and more diffusive discretization than that used for $\mathbf{f}(\mathbf{u})$ itself. Applying any approximate subdomain solver in an additive Schwarz manner tends to improve flop rates over the same preconditioner applied globally, since the smaller subdomain blocks maintain better cache residency, even apart from concurrency considerations [14]. Combining a Schwarz preconditioner with a Krylov iteration method inside an inexact Newton method leads to a synergistic, parallelizable nonlinear boundary value problem.

For purely elliptic problems, a single-level Schwarz preconditioner does not weak-scale due to degraded convergence. However, in applications, the convergence rate degradation is often not as serious as the scalar, linear elliptic theory would suggest. Its effects are mitigated using transient or pseudo-transient nonlinear continuation, which parabolizes the operator, given the diagonal dominance of the linear system to be inverted. Multilevel preconditioners are often preferred over single-level versions in aerodynamic and other applications; however recent strong scaling tests out to 32,768 cores of Yellowstone (29th-ranked in the Top500 at the time of writing) of the closely related unstructured CFD code NSU3D have shown that single-level solvers based on ILU-preconditioned GMRES, as in this study, can beat their multilevel cousins [15].

In the next section, we describe the main computational kernels in the NKS Solver within FUN3D, and describe their optimization challenges.

III. FUN3D Main Kernels and their Optimization Challenges

We look herein at the entire FUN3D application and at its most time consuming Newton-Krylov loop, which consists of the following four main types of kernels [3], [16]:

1) Edge-based "stencil op" loops: residual vector and Jacobian matrix evaluation and Jacobian-vector products
2) Sparse, narrow-band recurrences: approximate factorization and back-substitution
3) Vertex-based loops: state vector and auxiliary vector updates
4) Global collectives: vector inner products and norms

The majority of the execution time is expected to be spent in the edge-based loops (the "physics" of the application) and the recurrences (mostly linear algebra). Our profiling of the baseline code, described later in this paper, shows that these operations together account for 95% 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 subdomains of the partition. In the distributed memory context, edge-based loops are bound by the inter-node bandwidth if the latter does not scale with the architecture. Inner products are bound by the inter-node latency and network diameter. However, the shared-memory challenges are different, and are described in the next section for both edge-based loops and recurrences.

A. Edge-base loops

Edge-based "stencil op" loops: These loops predominantly occur in residual vector (flux) calculation, Jacobian matrix evaluation and Jacobian-vector products. Typically these loops have color-wise concurrency and local communication to complete the edges cut by the domain decomposition. A typical edge-based loop is schematically shown in Figure 1, where $\mathbf{G}$ is an arbitrary function. In FUN3D, these loops generate significant computational work per pair of vertices the edge operates upon, and comprise the majority of the floating-point operations of the code. These contribute to the bulk of the execution time. Therefore, these loops are one of the major focus areas of our study, and we present a number of shared-memory optimization techniques for these kernels later in this paper.

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.
3) Reducing the problem working set to fit into the on-die memory hierarchy.

With these overheads alleviated, these kernels are expected to scale with available computational power.

```
for all Edges do
    vertices forming the edge $[v_1, v_2]$
    $y[v1] \quad += \mathbf{G}(V[v1], V[v2])$
    $y[v2] \quad += \mathbf{G}(V[v2], V[v1])$
end for
```

Fig. 1: Edge-based loops

B. Sparse, narrow-band recurrences

Figure 2 schematically shows the sparse recurrences. The recurrences have limited parallelism, proportional to the number of independent edges. Based on the nonzero pattern, we can construct a task dependency graph of computing unknowns as shown in Figure 3b. In the dependency graph, the computation of each task roughly amounts to an inner product with length equal to the number of non-zeros in the corresponding matrix row. We can measure the parallelism available in a sparse triangular matrix as the ratio of the total number of floating point operations with the cumulative number of floating point operations in the longest dependency path.

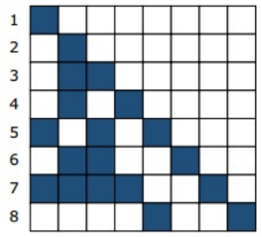
```
Input: factors $L$ and $U$, RHS vector $b$
Output: solution vector $x$
  // forward substitution
  for i = 1,2,...,n do
      $x(i) = b(i) - L(i, 1 : i-1) * [x(1), \ldots, x(i-1)]^T$
  end for
```

Fig. 2: Sparse recurrences

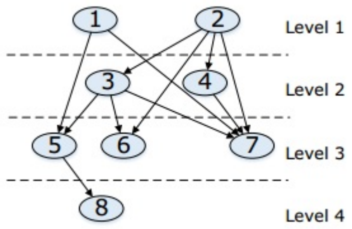
Further, the flop/byte ratio for these recurrences are usually low and these operations are expected to be bandwidth-bound. The primary challenges with these operations are:

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.
2) Load imbalance and synchronization overhead: Irregular sparsity pattern can result in load-imbalance, while limited parallelism can expose overhead of inter-core synchronization.

The block-CSR storage format (BCSR) is used for Jacobian matrix, with block size being the number of unknowns per vertex (4×4). It has been shown that BCSR has significant benefits, since it allows for coalesced loads (2 cache lines per block), reduces the index computation, and also alleviates the memory bandwidth pressure [2], [3]. The sparse matrix operations in PETSc are further optimized – the factored matrix is stored in the order it is accessed during the solve step and the diagonal blocks are additionally inverted within the ILU routine itself and then stored [17].



(a) Non-zero pattern of a lower triangular sparse matrix



(b) Corresponding dependency graph of forward solve

Fig. 3: Sparse recurrences

IV. Experimental Setup and Workload Characterization

In this section we describe our experimental platform, the CFD datasets used, and the performance profile of the non-optimized (baseline) FUN3D code.

A. *Platforms Used for Experiments*

For single node experiments we use a workstation with two Intel Xeon E5-2690 v2 processors (each processor has 10 cores), running at a clock speed of 3.0 GHz. The architecture features a super-scalar out-of-order micro-architecture supporting 2-way hyper-threading, resulting in the total of 20 hardware threads. 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 (AVX) [18]. In a single cycle, they can issue a 4-wide double-precision floating-point *multiply and add*, to two different pipelines. This allows for achieving full hardware utilization even when *multiply and add* can not be fused. Each core is backed by a 32 KB L1 and a 256 KB L2 cache, and all cores share an 24 MB last level L3 cache. Together, the 10 cores can deliver a peak performance of 240 Gflop/s of double-precision arithmetic using AVX. The system has 64 GB DDR3 memory. It consists of three channels running at 1600 MHz, which can deliver 42.2 GB/s of peak main memory bandwidth and STREAM bandwidth of 34.8 GB/s.

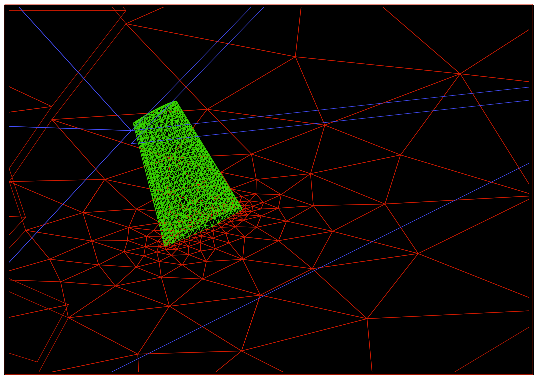


Fig. 4: Surface mesh over the ONERA M6 wing

We ran our scaling experiments up to 256 nodes of the Stampede supercomputer at the Texas Advanced Computing Center (TACC). Each node consists of two Intel Xeon E5-2680 processors and are interconnected with Mellanox InfiniBand FDR technology in a 2-level fat-tree topology. Each Intel Xeon E5-2680 processor has eight cores (with Hyper-threading disabled), with each core having 256KB private L1 caches and shares a common 30MB data cache with all other cores. All our experiments used the MVAPICH-1.9 MPI library and the MPIP-3.0.3 profiling tool for generating MPI statistics.

B. *CFD Datasets*

We use the surface mesh over the ONERA M6 wing, shown in Figure 4. This a classic CFD validation case for external flows because of its simple geometry combined with complexities of transonic flow [19]. It has almost become a standard for CFD codes because of its inclusion as a validation case in numerous CFD papers over the years.

For this geometry, we use two different meshes – the two largest meshes of the 1999 study [2], previously referred to as Mesh-C and Mesh-D (listed in Table I). Mesh-C with 2.4 million edges is representative of the problem size solved on a single compute node and we have used this for all the single-node experiments. Mesh-D is the larger mesh with close to 19 million edges, and we have used this for all the multi-node experiments. The unstructured mesh used here requires explicit storage of neighborhood information.

	Mesh-C	Mesh-D
Vertices	$3.58e5$	$2.76e6$
Edges	$2.40e6$	$1.89e7$
Time steps	13	29
Linear iterations	383	1709
Execution Time (s)	$2.82e2$	$1.02e4$

TABLE I: Baseline Performance

C. *Baseline Performance Profile*

As our baseline, we use the single threaded performance the original PETSc-FUN3D code [2], [3], [20] compiled out-of-the box and measured on the test platform. The reason for using single threaded baseline was twofolds: 1. The out-of-the-box code did not have threading included, and we have included our threading strategies as one of the single-node

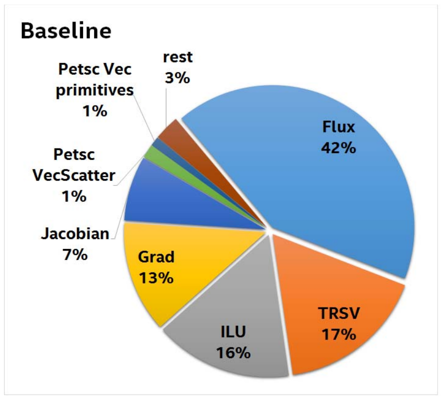


Fig. 5: Performance profile of Base Application

optimizations strategies evaluated in this paper, and 2. The MPI-results are included in the multi-node results section (Section VI.B). Table I lists the number of time steps, iterations and the execution time to converge for the baseline code for both Mesh-C and Mesh-D. Figure 5 shows the performance profile of the baseline code on a single node. The primary kernels and their contributions to the total execution time are: flux computation (42%), triangular direct solver (TRSV / Matsolve 17%), incomplete LU decomposition of the Jacobian matrix (16%), gradient calculation (13%), and construction of the Jacobian matrix (7%). These kernels together contribute about 95% of the total execution time and hence are the primary focus of the current work.

Having identified the kernels that constitute the compute hotspots, in the next section we describe the key optimizations and performance analysis for these kernels.

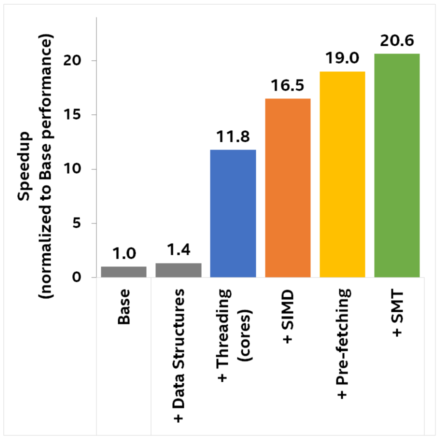
V. Shared Memory Kernel-level Optimizations and Performance Results

In this section, we describe various shared-memory optimization strategies as well as kernel-wise optimizations. We focus on edge-based loops and sparse recurrences, since these are the computational patterns contributing to the majority of the execution time.

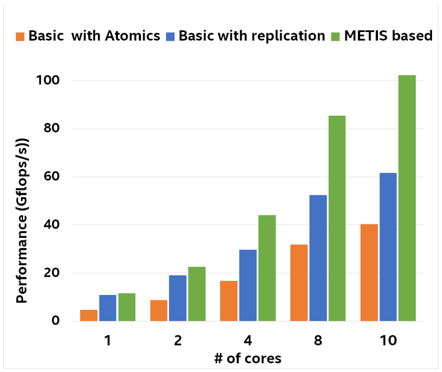
A. *Edge-based loops*

The flux kernel is used as a representative example to detail the optimizations for the edge-based loops. As seen from the performance profile shown in Figure 5, the flux kernel contributes a significant fraction of the execution time. The edge-based loops are essentially stencil operations across each edge, computing at vertices associated with an edge. Since each vertex is involved in many discrete stencil operations, this results in a fairly high ratio of work per data-size.

For the flux kernel this ratio is 9.4 flops per byte of data accessed. This calculation traverses all the edges in a subdomain, updating only the local vertices. Kaushik *et al.* [4] have done initial evaluations of parallelizing the flux kernel on the IBM Blue Gene/P machine, but this study was limited to only 4 OpenMP threads per rank. The conclusion of this work was that manual partition of the subdomain using METIS [21] is the best approach to partition the processing of edges across multiple threads. In the current work, in addition to re-evaluating these claims for increased number of threads and we have also evaluated alternative parallelization strategies as well as explored SIMD parallelism, which is a critical optimization on modern day multi- and many-core architectures. The vertex numbering is reordered using Reverse Cuthill-McKee (RCM) algorithm [22] to improve locality. Furthermore, the vertices at one end of each edge are sorted in an increasing order to make the access pattern more regular. Figure 6a shows the speed-ups due to these optimizations on the flux kernel. As seen from the figure, the performance of optimized flux kernel is 20.6X over the base (sequential) code for 10 cores (20 threads). Figure 6b shows the scaling with number of cores for different partitioning strategies. Since the edge-based loops are predominantly compute-bound, the performance scales almost linearly with the number of cores.



(a) Flux kernel: Speed-ups due to various optimizations



(b) Flux kernel: Speed-ups with different parallelization strategies

Fig. 6: Performance results for flux kernel

The contribution of different optimizations to this overall speedup is explained below.

- *Threading*: The edge-base loops have color-wise concurrency, i.e., edges that do not share the same vertices can be processed in parallel. However coloring-based partitioning of an unstructured mesh results in sub-optimal spatial locality among the concurrently processed edges [15]. Hence we restrict ourselves to domain-decomposition based parallelization strategies even within a node for edge loops. We evaluate three different methods to extract parallelism. First, we divide edges in natural order between threads and use an atomic update to handle the dependency between threads processing edges which share vertices. This strategy is referred to as "Basic partitioning with atomics". It can been seen from Figure 6b that even though this strategy scales almost linearly with the number of cores, the performance is low. This is primarily due to the overhead of the atomic update. We further divide the vertices also amongst the threads (based on natural order) in addition to partitioning the edges, with replication. For multiple threads handling edges which share a vertex, only the thread which owns that vertex is responsible for processing it ("owner-only writes"). We refer to this strategy as "Basic partitioning with replication". This avoids write contention between threads and hence the vertices can be updated without atomics. In this case, as expected, the absolute performance is better than with atomics but it doesn't scale as well with increasing number of cores. This can be attributed to the load imbalance between threads and overheads due to replication. Due to the replication of edges across partitions, there is significant redundant compute; with 20 threads (10 cores) the natural order based splitting of vertices results in a staggering 41% increase in compute. Using METIS to partition the nodes across the threads improves the balance of work between threads and reduces the edge-replication across partitions. This is referred to as "METIS based partitioning". This strategy, in addition to giving higher absolute performance, also scales almost linearly with the number of cores. With METIS, the overhead due to replication with 20 threads (10 cores) is reduced to a nominal 4%. Even with METIS, this overhead is expected to be significant with increased parallelism on emerging many-core architectures. For instance, our initial experiments with 240 threads on a many-core architecture indicate that this overhead becomes as high as 15%.

- *Data structures*: As each thread processes consecutive edges, the accesses to the edge data is ordered and regular (streaming). To facilitate better reuse and easier vectorization, the edge data is stored as a Structure of Array (SoA) data structure. Accessing the vertex data requires gathering data from non-consecutive vertices associated with an edge. However, within a vertex the variables (states, gradients, geometric variables) are used almost successively and hence the node data is stored as multiple Array of Structures (AoS) data structures. That is, state variable are clubbed together ($nVertices \times 4$), the gradient in each of the three dimensions for these state variables ($nVertices \times 4 \times 3$) and then geometrical values ($nVertices \times 3$). Regardless of which format we use for node data, accesses to node data will require irregular accesses and hence can not be done with the vector load. For architectures without gather support, storing data in SoA format results in issuing 4 sequential loads, one per field, to fill an 8-wide SIMD register. However, when node data is stored in AoS format, consecutive state variables can be loaded into SIMD registers using a vector load, one per node. Gathers can be performed via register permutation out of the register file. Overall, AoS-based approach requires fewer loads than SOA-based approach and it better utilizes available issue ports, thus being more efficient as well. Detailed cache analysis indicate that this results in a 20% better reuse across L1 and L2 caches, which translates into a 40% performance benefit (Figure 6a).
- *Exploring SIMD*: We vectorize across edges, with each SIMD lane used for processing an edge entirely. Thus, we have four edges being processed concurrently within a thread. To address the possible dependency across these edges, we separate the write-out part from the compute. The dependency is eliminated for the compute, with the computed values stored 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 (less than 5%), since it is amortized by large amount of compute in the flux kernel. With this restructuring of the kernel, the code generated by the Intel auto-vectorizing compiler is comparable to the hand-coded SIMD intrinsics. However, we observe that the auto vectorization performs slightly better, because the compiler generates more optimized code for gathers (node-data). Overall, SIMD-optimization result in a 40% improvement in performance (Figure 6a).
- *Software Prefetching*: Since this is an unstructured mesh, the sequence of nodes associated with successive edges does not follow a regular order. However the nodes associated with each mesh are known *a priori*. We use this information to prefetch the node and edge data for successive edges using software prefetch instructions to both L1 and L2 caches. With careful tuning of these prefetch instructions the execution time reduces by 28% which translates into a 15% performance benefit (Figure 6a).

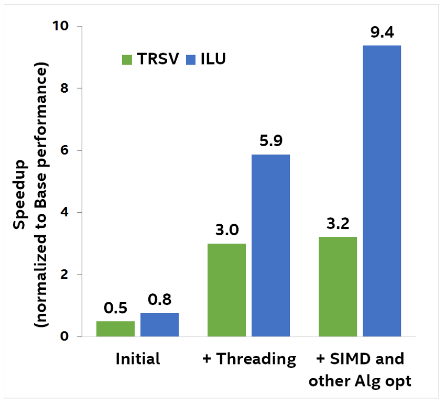
B. Sparse, narrow-band recurrences

Incomplete LU (ILU) decomposition and triangular solver (TRSV) are the two important kernels under this category. Figure 7a shows the speed-ups achieved for these kernels with different optimization strategies, and Figure 7b shows the performance in terms of achieved bandwidth for different parallelization strategies.

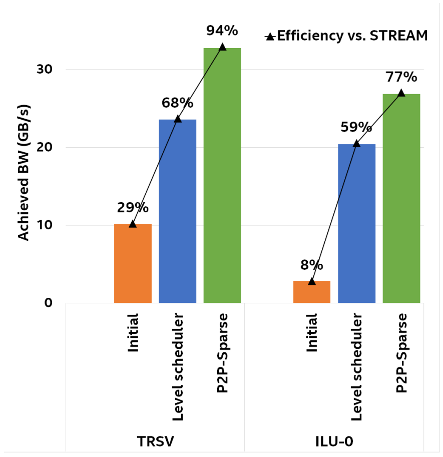
The ILU decomposition of the Jacobian matrix is done once every time step in our tests. (Reusing the factors even as the Jacobian evolves underneath is a problem-dependent optimization that is worth pursuing in practice.) The iterative solver uses the factored matrix as a preconditioner every linear iteration. With optimization the ILU kernel achieves a speedup of 9.4X and the sparse triangular solver a speed up of 3.2X with 10 cores (20 threads) over the base (sequential) code. Both these kernels are bandwidth-bound owing to low flop/byte ratio. The (sparse) Jacobian matrix is constructed as a blocked-diagonally dominant matrix with 4×4 blocks and uses the BSR sparse representation.

For TRSV described in Figure 2, the primary compute is multiplying a 4×4 matrix with a 4×1 vector for each non-zero block. There is no reuse across blocks (streaming access). The TRSV kernel achieves 94% of the STREAM bandwidth on a single node with 20 threads (10 cores), as shown in Figure 7b. We observe that the bandwidth starts to saturate beyond 4 cores.

The ILU factorization involves higher compute, since for each block, the non-zero blocks in that particular row/column prior to it need to processed [23]. The primary computation here is a 4×4 matrix-matrix multiplication and inversion of the diagonal block for each row. Here the performance scales up to 8 cores, beyond which the kernel is bandwidth-bound.



(a) Performance Optimizations



(b) Parallelization strategies

Fig. 7: ILU and TRSV

The access pattern with this kernel is non-regular and hence achieved bandwidth efficiency is not as high the TRSV kernel.

The available parallelism is limited with these recurrences. We evaluate the following two strategies to extract the parallelism for these operations: (1) Level-scheduling with barriers: executes task graphs in 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. Since tasks in the same level are independent, they can execute in parallel [24], [25]. This has issues of load imbalance across threads since amount of work with successive levels tends to decrease drastically. (2) Sparsification with approximate transitive edge reduction (P2P-Sparse) by Park *et al.* [26], which sparsifies (i.e., eliminates) the redundant dependencies with point-to-point synchronization and improves scalability. Even though these techniques have been applied primarily for the sparse triangular solver, they are also applicable to the ILU since both these operate on same sparsity pattern.

- *Threading*: Figure 7b compares the performance of both the ILU and TRSV kernels with the two parallelizing strategies of Level scheduler and P2P-Sparse. Clearly the sparsification of synchronization approach performs better both in terms of absolute performance and also scaling across cores for both the kernels. With increasing parallelism, the benefit from the sparsification approach is expected to further increase.
- *Algorithmic optimization*: The ILU decomposition requires maintaining a temporary buffer typically the size of number of rows to store the partial results for each row. With general sparse matrices the number of rows/columns accessed for processing each row is orders of magnitude smaller than the total number of rows, even though distance between these rows/columns might be much higher. The large temporary buffer not only results in increased access stride, but also in increased working set with threading. We circumvent this issue by using a static access information to map the accesses to a reduced (compressed) temporary buffer. Again, this becomes critical with higher number of threads.
- *Exploring SIMD*: The key computations are with small (4×4) matrices and vectorization is done within a block. To be able to effectively utilize the available SIMD units, we manually vectorize these operation with vector intrinsics. Since these kernels are primarily bandwidth-bound, the performance benefits with vectorization are not very significant.

VI. Full Application Performance Results for Single- and Multiple Nodes

In this section we describe the single- as well as multi-node performance results for the FUN3D application. We have retained all the single-node optimizations – interlacing, blocking and reordering – previously used by W. Gropp *et al.* [3] for the FUN3D code. In addition, we combined all our kernel-wise optimizations described in the previous section.

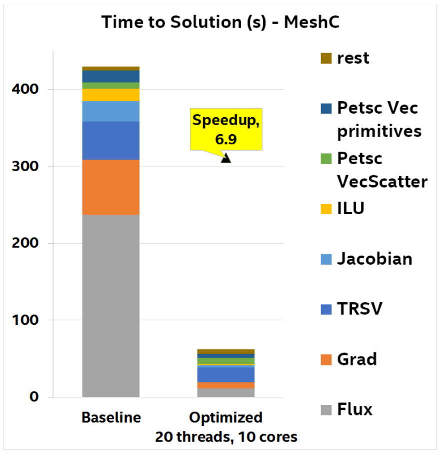
A. Single Node results

Figure 8a shows the "time to solution" comparison for the single-node optimized code with the base sequential code. We achieve a 6.9X speedup for the full application on Intel Xeon E5 2690 processor (10 cores). The figure also shows speed-ups achieved for various kernels.

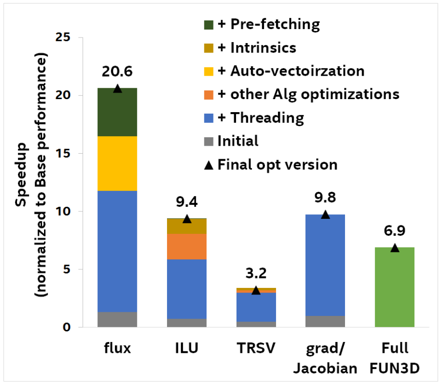
The edge-based loops are expected to be compute bound and we see this behavior with the flux, Grad and Jacobian kernels, with the performance scaling (almost) linearly with the number of cores. For the flux kernel, which is the primary compute hot-spot, the additional optimizations result in 2X performance benefit. Whereas the sparse triangular solver (TRSV) and ILU are bandwidth-bound and scales only with the bandwidth utilization per core.

The sparse triangular solver (TRSV) becomes the primary hot-spot post-optimization. Owing to the fact that this is bandwidth-bound, the speedup efficiency for the full application is limited to 69% (6.9X on 10 cores). With the contribution of the optimized primary kernels dropping, the ‘other’ auxiliary operations become quite significant, contributing to about 30% of execution time. Of the remaining operations, the major contribution is from the vector primitives (VecMAXPY, VecWAXPY, VecMDOT, etc.) and the vector scatter operations (VecScatter), which are PETSc native functions. We have further optimized the vector primitives by either replacing them with our own optimized (multi-threaded, vectorized) implementations or using the corresponding MKL functions.

Since our optimizations are all performed within the PETSc framework, it allows seamless use of the plethora of algorithms available in PETSc. For example, we evaluated our



(a) Optimized PETSc-FUN3D Performance



(b) Kernel-wise performance speedup

Fig. 8: Optimized FUN3D application

optimizations for preconditioner with fill-in and compared the performance for preconditioner without fill-in, and studied their parallelization potentials and performance gains.

The original version of the PETSc-FUN3D for the Schwarz preconditioner uses an incomplete LU decomposition with fill level of 1 (ILU-1) for the Jacobian matrix. This was found to be optimal from the algorithm perspective [23], [27], as it offers faster convergence than the ILU without fill-in (ILU-0). The previous study was limited to considering the trade-off due to increase in work with fill-in and reduction in number of iterations to converge. Level of fill-in also affects the available parallelism. As described in section III, the parallelism is measured as the ratio of total number of floating point operations by the cumulative number of floating point operations in the longest dependency path of a task dependency graph based on the non-zero pattern. With increase in non-zeros with fill-in the available parallelism drastically reduces.

Table II compares the amount of parallelism, number of iterations to converge, and the speed-up on 10 cores for FUN3D application using ILU-0 and ILU-1. As seen from the table, ILU-1 offers faster convergence but it offers less parallelism, and this characteristic shows interesting results when parallelized to 10 cores. Due to more inherent parallelism in ILU-0, although it takes more iterations to converge, it starts scaling better with the number of cores, and at 10 cores, ILU-0 outperforms ILU-1 by about 1.3X.

	Mesh-C	
	ILU-0	ILU-1
Available parallelism	$248X$	$60X$
Linear iterations	777	383
Execution Time (s) on single core	430	282
Execution Time (s) on 10 cores	62	81
Speed-up over single core	$6.9X$	$3.5X$

TABLE II: Comparison of ILU-0 and ILU-1 for parallelism and performance

B. Multi-node Results

There are two primary objectives for this section. The first objective is to demonstrate that the shared memory optimizations presented in the previous section continue to provide similar performance benefits as we scale to larger number of nodes. Our second objective is to characterize Krylov solvers for their scaling properties and establish the scaling limits. Further, we examine benefits of thread-level parallelism within a node.

1) Scaling studies with cache and SIMD optimizations: Here, we compare the results of the baseline PETSc version against an optimized version with the cache- and SIMD-optimizations included, in a strong scaling experiment. The specific versions used are labeled as: *Baseline* code running sixteen MPI processes per node (one per core), *Optimized* code running sixteen MPI processes per node (one per core) with cache- and SIMD-optimizations.

Figure 9 shows the execution times for the FUN3D application as we scale it to 256 nodes of the Stampede supercomputer at TACC. The figure shows that cache- and SIMD-optimizations result in higher performance compared to the baseline version. This performance benefit is seen at all scales, with the execution time speedups ranging from about 16% to about 28%. *Thus, it is clear that the cache- and SIMD-optimizations we discussed in this paper are essential for improving the performance and continue to give benefit as we scale the application to multiple nodes.*

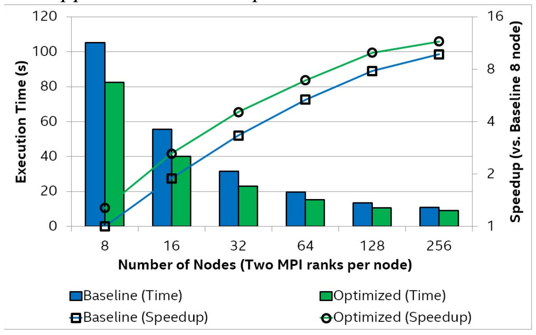


Fig. 9: Scaling of FUN3D application to 256 nodes

2) Characterization of Krylov solvers scaling on multi-node: As we increase the number of nodes, communication starts to dominate the performance and the largest mesh (Mesh-D) becomes communication bound at 256 nodes (with communication time being 70% of the total execution time). The majority ($90\%+$) of the communication overhead is due to the MPI_Allreduce operations in the Krylov solver. Point-to-point messages contribute less than 5% of the communication overhead. The datasets used in [2] become communication bound much sooner, as early as 256 nodes on current highly parallel architectures. Further, with special HW support for global collectives as in the IBM Blue Gene/P machine, the same dataset (Mesh-D) scales only up to 1024 nodes [4]. The global collectives which are inherent to the Krylov solver become the primary scaling bottleneck, thereby limiting its applicability at large scales. There are recent algorithmic developments [28], [29] to circumvent this limitation. We are further evaluating these improved Krylov solvers as future work.

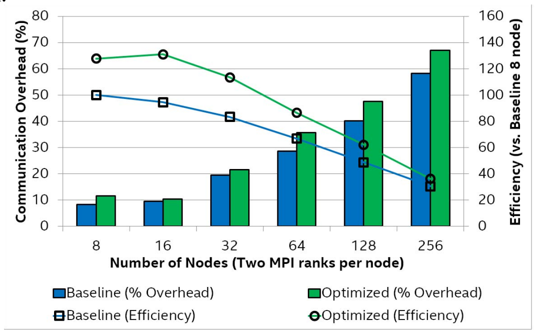


Fig. 10: Communication overheads in scaling of FUN3D

3) Thread-level parallelism studies: Modern architectures are trending towards multi- and many-cores, offering an increasing degree of thread-level parallelism within each node. Therefore, it is imperative for any large-scale application to exploit this multi-level parallelism. In this study, we use a hybrid OpenMP+MPI strategy to exploit this parallelism, and examine whether our shared-memory optimizations continue to give benefits as we scale.

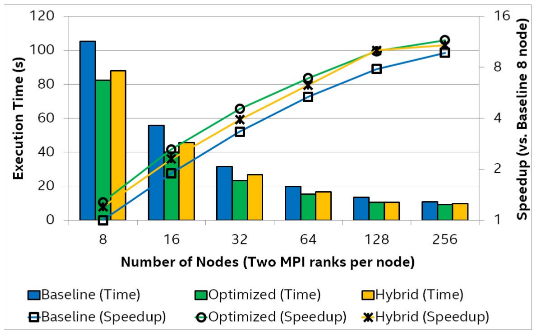


Fig. 11: Baseline, Optimized and Hybrid versions scaled to 256 nodes

Figure 11 shows the comparison of execution times and speedups of the "Baseline" (Section VI-B1), "Optimized with cache- and SIMD-optimizations" (Section VI-B1), and "Hybrid" version. The "Hybrid" version is defined as "PETSC FUN3D code with all shared memory optimizations (cache, SIMD, threading) enabled running two MPI processes per node (one per socket), with each MPI processes using eight threads (one per core)". While Optimized and Hybrid include the same SIMD and cache optimizations, the difference is that the "Hybrid" version additionally includes threading optimizations and hence they have different number of threads per rank and number of ranks per socket.

The Hybrid version performs better than the Baseline by around 10% to 23%. This demonstrates the performance benefits of cache, SIMD, and threading optimizations. However, these benefits are lower than that those obtained by the "Optimized" version (i.e., MPI-only with cache- and SIMD-optimizations). The coarse-grained explicit parallelization of the MPI-only implementation parallelizes most portions of the PETSc library routines, while the Hybrid version introduces thread-level parallelism only for the FUN3D kernels. This lack of thread-level parallelism within the PETSc library for certain vector and communication primitives (such as VecNorm, VecMDot, VecScatterEnd) increases the Amdahl's fraction for the "Hybrid" case. However the MPI-only parallelization poses its own set of challenges/limitations – increase in the number of sub-domains results in the degradation of the convergence properties due to the reduced coupling and we observe up to 30% increase in iterations at 256 nodes and increased pressure on the resources within a node. Hence, with increased parallelism within a node we expect the "Hybrid" version to perform better. Therefore, our call to the PETSc development community is to optimize/parallelize the native routines, which would enable better utilization of modern architectures with higher degree of thread-level parallelism. Pending these optimizations, our analysis shows that using our cache- and SIMD-optimization together with MPI-only implementation is the fastest of the three examined approaches.

VII. Conclusions and Future Work

Computational aerodynamics is an important workload for high performance computing, and is often used as a representative workload in benchmarking and tuning hardware and software programming environments. Most of the past work by the scientific community in the space of scaling of CFD applications has been focused on weak scaling to tens of thousands of processors, and corresponding distributed-memory optimizations. With the advent of modern multi- and many-core processors with high degree of fine-grained thread parallelism available within a single shared-memory node, practitioners are now forced to look to strong scaling with the advent of massive thread parallelism within a single shared-memory node. In this work, we present analysis and optimization of an incompressible implicit flow solver on modern parallel systems. We study the flow over an aircraft wing geometry meshed using unstructured grid. Unstructured grids pose further challenges to optimization primarily due to the severe lack of spatial locality and highly nonuniform access patterns.

We chose the well characterized application PETSc-FUN3D as our case study application. We demonstrate several optimization techniques for shared-memory systems, including threading, data structure optimizations, vectorizations, software prefetching, etc. We break the application down into various kernels as well as computational patterns such as edge-based loops and sparse recurrences, and demonstrate optimizations for these characteristics patterns. Our results from aerodynamics should bear on PDE-based workloads from reservoirs, geophysics, materials processing, solid mechanics, reacting flows, magnetohydrodynamics, and many other temporally stiff problems posed on geometrically irregular or adaptively refined domains, which shares many of the key algorithmic characteristics with our PETSc-FUN3D code.

Our optimizations show a 6.9X speed-up in performance on a single node Intel Xeon E5 2690 processor (10 cores) relative to out-of-the-box compilation. Further, we also perform scaling studies on multi-node clusters and show that our shared memory optimizations continue to provide performance benefits as we scale to larger number of nodes. Our experiments on 256 nodes Stampede supercomputer at Texas Advanced Computing Center (TACC) show that cache- and SIMD-optimizations result in higher performance at all scales, compared to the baseline case without these optimizations.

Most of our shared-memory optimizations are expected to extend to modern many-core architectures such as Intel® Xeon Phi™ Coprocessor architecture. Thus, this work is the precursor for our ongoing work to study the implication of increasing parallelism within a single node with emerging multi- and many-core architectures. Our work has shown that certain library routines in PETSc (for example, vector primitives) are not thread-level parallelized, and these become bottlenecks preventing PETSc from exploiting full benefits for multi-level parallelism available in today's architecture. Optimizations of these routines by the PETSc community would have a major impact on performance gains on modern multi- and many-core architectures.

In future, we plan to optimize FUN3D on many-core architectures, and also study the multi-node scaling characteristics on a much large cluster of several thousands of nodes, and using much larger data sets including a full aircraft configuration (including the fuselage).

Acknowledgment

The authors would like to thank Jed Brown, Barry Smith, Matt Knepley, Hong Zhang and others from the PETSc team for their support. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. The authors would also like to thank Prof. Juan Alonso of Stanford University for his help in proof-reading this paper.

References

[1] A. C. Duffy, D. P. Hammond, and E. J. Nielsen, "Production level CFD code acceleration for hybrid many-core architectures," NASA/TM-2012-217770, October, Tech. Rep., 2012.

[2] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Achieving high sustained performance in an unstructured mesh CFD application," in *Supercomputing*. ACM, 1999, p. 69.

[3] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "High-performance parallel implicit CFD," *Parallel Computing*, vol. 27, no. 4, pp. 337–362, 2001.

[4] D. Kaushik, D. Keyes, S. Balay, and B. Smith, "Hybrid programming model for implicit PDE simulations on multicore architectures," in *OpenMP in the Petascale Era*. Springer, 2011, pp. 12–21.

[5] S. Balay, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, "PETSc Web page," 2014. [Online]. Available: http://www.mcs.anl.gov/petsc

[6] ——, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.4, 2013. [Online]. Available: http://www.mcs.anl.gov/petsc

[7] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[8] W. K. Anderson and D. L. Bonhaus, "An implicit upwind algorithm for computing turbulent flows on unstructured grids," *Computers and Fluids*, vol. 23, pp. 1–21, 1994.

[9] W. K. Anderson, R. D. Rausch, and D. L. Bonhaus, "Implicit/multigrid algorithms for incompressible turbulent flows on unstructured grids," *Journal of Computational Physics*, vol. 128, pp. 391–408, 1996.

[10] P. Roe, "Approximate Riemann solvers, parameter vectors, and difference schemes," *Journal of Computational Physics*, vol. 43, pp. 357–372, 1981.

[11] W. Mulder and B. Van Leer, "Experiments with implicit upwind methods for the Euler equations," *Journal of Computational Physics*, vol. 59, pp. 232–246, 1985.

[12] D. A. Knoll and D. E. Keyes, "Jacobian-free Newton-Krylov methods: A survey of approaches and applications," *Journal of Computational Physics*, vol. 193, pp. 357–397, 2004.

[13] A. Toselli and O. B. Widlund, *Domain Decomposition Methods: Algorithms and Theory*. Springer, 2005, Springer Series in Computational Mathematics, vol. 34.

[14] G. Wang and D. K. Tafti, "Performance enhancements on microprocessors with hierarchical memory systems for solving large sparse linear systems," *International Journal for High Performance Computing Applications*, vol. 13, pp. 63–79, 1999.

[15] D. Mavriplis and K. Mani, "Unstructured mesh solution techniques using the NSU3D solver," 2014, aIAA Paper 2014-0081, Presented at the 52nd AIAA Aerospace Sciences Conference, National Harbor, MD.

[16] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Performance modeling and tuning of an unstructured mesh CFD application," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, pp. 34–34.

[17] B. Smith and H. Zhang, "Sparse triangular solves for ILU revisited: data layout crucial to better performance," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 386–391, 2011.

[18] "Intel® architecture instruction set extensions programming reference," 2014.

[19] V. Schmitt and F. Charpin, "Pressure distributions on the ONERA-M6 wing at transonic mach numbers," *Experimental data base for computer program assessment*, vol. 4, 1979.

[20] S. Bhowmick, D. Kaushik, L. McInnes, B. Norris, and P. Raghavan, "Parallel adaptive solvers in compressible PETSc-FUN3D simulations," in *Proceedings of the 17th International Conference on Parallel CFD*, 2005.

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[22] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*. ACM, 1969, pp. 157–172.

[23] E. Chow and Y. Saad, "Experimental study of ILU preconditioners for indefinite matrices," *Journal of Computational and Applied Mathematics*, vol. 86, no. 2, pp. 387–414, 1997.

[24] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal of High Speed Computing*, vol. 1, no. 01, pp. 73–95, 1989.

[25] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," NVIDIA Technical Report, NVR-2011-001, Tech. Rep., 2011.

[26] J. Park, M. Smelyanskiy, and P. Dubey, "Sparsifying synchronizations for high-performance shared-memory sparse triangular solver," in *International Supercomputing Conference (ISC)*, 2014.

[27] A. Chapman, Y. Saad, and L. Wigton, "High order ILU preconditioners for CFD problems," *UMSI research report/University of Minnesota (Minneapolis, Mn). Supercomputer institute*, vol. 96, p. 14, 1996.

[28] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose, "Hiding global communication latency in the GMRES algorithm on massively parallel machines," *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. C48–C71, 2013.

[29] L. C. McInnes, B. Smith, H. Zhang, and R. T. Mills, "Hierarchical Krylov and nested Krylov methods for extreme-scale computing," *Parallel Computing*, vol. 40, no. 1, pp. 17–31, 2014.