Reducing Communication Overhead in Large Eddy Simulation of Jet Engine Noise

Yingchong Situ* Lixia Liu* Chandra S. Martha† Matthew E. Louis†
Zhiyuan Li* Ahmed H. Sameh* Gregory A. Blaisdell† Anastasios S. Lyrintzis†
*Department of Computer Science and †School of Aeronautics and Astronautics
Purdue University
West Lafayette, United States
{ysitu, liulixia, cmartha, louism, zhiyuanli, sameh, blaisdel, lyrintzi}@purdue.edu

***Abstract*—Computational aeroacoustics (CAA) has emerged as a tool to complement theoretical and experimental approaches for robust and accurate prediction of sound levels from aircraft airframes and engines. CAA, unlike computational fluid dynamics (CFD), involves the accurate prediction of small-amplitude acoustic fluctuations and their correct propagation to the far field. In that respect, CAA poses significant challenges for researchers because the computational scheme should have high accuracy, good spectral resolution, and low dispersion and diffusion errors. A high-order compact finite difference scheme, which is implicit in space, can be used for such simulations because it fulfills the requirements for CAA. Usually, this method is parallelized using a transposition scheme; however, that approach has a high communication overhead. In this paper, we discuss the use of a parallel tridiagonal linear system solver based on the truncated SPIKE algorithm for reducing the communication overhead in our large eddy simulations. We report experimental results collected on two parallel computing platforms.**

***Keywords*-finite difference methods; iterative solution techniques; linear systems; numerical algorithms; parallel algorithms**

I. Introduction

Computational aeroacoustics (CAA) has emerged as a relatively new discipline and a robust and accurate tool that complements traditional theoretical and experimental approaches in the prediction of sound levels from aircraft airframes and engines. CAA, unlike the related discipline of computational fluid dynamics (CFD), involves the accurate prediction of small-amplitude acoustic fluctuations and their correct propagation to the far field. In that respect, CAA poses significant challenges for researchers because the computational scheme should have high accuracy, good spectral resolution, and low dispersion and diffusion errors.

The state of the art of CAA prediction of far-field noise is based on time-dependent simulation of noise-generating turbulent flows coupled with integral methods for propagating the noise to the observer location. The highest level of simulation, based on the Navier-Stokes equations, is direct numerical simulation (DNS), in which time-dependent motions of all revelant length scales are resolved directly without using any turbulence model. While theoretically DNS can deliver the best accuracy in numerical results among mainstream methodologies for numerical CAA simulation, it suffers from the major drawback that its computational cost is infeasible for turbulent flows of Reynolds numbers of practical engineering interest. At the opposite extreme in simulation philosophy to DNS, Reynolds-averaged Navier-Stokes (RANS) equations model the full range of time-dependent motions of all length scales using turbulence models. In comparison with DNS, RANS significantly reduces the computational cost, but only at the expense of the flow physics. Large eddy simulation (LES) strikes a balance and is a compromise between DNS and RANS; it directly resolves eddies larger than the grid scale and captures the effect of small eddies using a subgrid-scale model. Such a methodology allows LES to use, in comparison with DNS, a coarser grid that is fine-grained just enough to resolve large eddies, maintaining the feasibility of simulating turbulent flows at high Reynolds numbers; meanwhile, the subgrid-scale model ensures that the influences of small eddies are retained even though they are not directly simulated.

Lying at the core of numerical methods for three-dimensional LES are spatial differentiation of flow variables, and spatial filtering for suppressing numerical artifacts. Both of these operations involve solving tridiagonal linear systems along the three axis directions of the computational space. In our previous efforts in developing code for three-dimensional LES, we used a transposition scheme [1] where the computational space is transposed as necessary so that all data for each individual system are available to a single processor. This allowed us to utilize the tridiagonal linear system solver in LAPACK [2] to attain high accuracy as well as high efficiency and achieve almost perfect scalability in our previous performance experiments.

Unfortunately, as computing platforms evolve, and the gap between processor speed and interconnection network bandwidth further widens, in our most recent experiments, the high communication overhead inherent to the transposition scheme exerted significant impact on its parallel performance and severely limited its efficiency. Also, the transposition scheme limits the number of processors used to be no more than the number of planes in a given direction, when a one-dimensional partitioning is done. This prompted us to investigate alternatives to the transposition scheme. Among a multitude of possible choices, we choose to get rid of transposition of the computational space by employing a

Table I
MEANINGS OF SYMBOLS IN (1)

Symbol	Meaning
t	Time
ξ, η, ζ	Generalized curvilinear coordinates in computational space
J	Jacobian that maps physical space to computational space
$\boldsymbol{Q}$	Vector of conservative flow variables (density, three components of momentum, and total energy)
$\boldsymbol{F}, \boldsymbol{G}, \boldsymbol{H}$	Inviscid flux vectors
$\boldsymbol{F_v}, \boldsymbol{G_v}, \boldsymbol{H_v}$	Viscid flux vectors

new parallel tridiagonal linear system solver based on the truncated SPIKE algorithm [3] with the primary design goal of significantly reducing the communication overhead. The process and effects of applying the new solver are the subjects of this study.

To present the performance of our new LES implementation in comparison with the original implementation, we organize the rest of this paper as follows. Section II briefly explains the numerical methods employed in our large eddy simulations and illustrates the inner workings of our original LES implementation. Section III presents details of the SPIKE-based tridiagonal linear system solver. In particular, we elaborate on specific optimizations aimed at reducing the communication overhead. Section IV presents the experimental performance data of the new LES implementation. Section V concludes the paper and looks at future work.

II. NUMERICAL METHODS AND EXISTING IMPLEMENTATION OF 3D LES

A. Physical and Mathematical Foundations

The essense of large eddy simulation is to solve a system of Favre-filtered unsteady, compressible, nondimensionalized Navier-Stokes equations formulated in conservative form [1]. The system is succinctly captured as

$$\frac{1}{J}\frac{\partial \boldsymbol{Q}}{\partial t} + \frac{\partial}{\partial \xi}\left(\frac{\boldsymbol{F}-\boldsymbol{F_v}}{J}\right) + \frac{\partial}{\partial \eta}\left(\frac{\boldsymbol{G}-\boldsymbol{G_v}}{J}\right) + \frac{\partial}{\partial \zeta}\left(\frac{\boldsymbol{H}-\boldsymbol{H_v}}{J}\right) = \boldsymbol{0}. \tag{1}$$

The meanings of the symbols in (1) are listed in Table I. To discretize the problem, we represent physical space using a three-dimensional curvilinear grid, operate in discrete time steps and subsequently replace partial differentiation in (1) with a partial difference. To simplify the numerical scheme, we map physical space to a computational space represented by a three-dimensional Cartesian grid. The mapping is captured by the Jacobian J.

We rewrite (1) as

$$\frac{\partial \boldsymbol{Q}}{\partial t} = \mathbf{RHS}(\boldsymbol{Q}; t) \tag{2}$$

where

$$\mathbf{RHS}(\boldsymbol{Q}; t) = -J\left(\frac{\partial}{\partial \xi}\left(\frac{\boldsymbol{F}-\boldsymbol{F_v}}{J}\right) + \frac{\partial}{\partial \eta}\left(\frac{\boldsymbol{G}-\boldsymbol{G_v}}{J}\right) + \frac{\partial}{\partial \zeta}\left(\frac{\boldsymbol{H}-\boldsymbol{H_v}}{J}\right)\right) \tag{3}$$

and use the classical fourth-order four-step explicit Runge-Kutta scheme to perform time integration of (2). For each iteration of time integration, $\mathbf{RHS}(\boldsymbol{Q}; t)$ is evaluated multiple times with different $\boldsymbol{Q}$ and t. Details of evaluating $\mathbf{RHS}(\boldsymbol{Q}; t)$ are given in [1].

Evaluation of $\mathbf{RHS}(\boldsymbol{Q}; t)$ involves intensive spatial partial differentiation tasks along the ξ-, η- and ζ-directions apart from pointwise computation. To compute the spatial derivatives at interior grid points away from boundaries, we follow [4] in employing the following nondissipative sixth-order compact scheme suggested in [5]:

$$\frac{1}{3}f'_{i-1} + f'_i + \frac{1}{3}f'_{i+1} = \frac{7}{9\Delta\xi}(f_{i+1} - f_{i-1}) + \frac{1}{36\Delta\xi}(f_{i+2} - f_{i-2}) \tag{4}$$

where f'_i is the approximation of the first derivative of function f at point i along the ξ-direction. For near-boundary points at $i = 1, 2$, we apply the following third-order one-sided compact scheme and fourth-order central compact scheme [1]:

$$f'_1 + 2f'_2 = \frac{1}{2\Delta\xi}(-5f_1 + 4f_2 + f_3), \tag{5a}$$

$$\frac{1}{4}f'_1 + f'_2 + \frac{1}{4}f'_3 = \frac{3}{4\Delta\xi}(f_3 - f_1). \tag{5b}$$

Similar formulations are applied for boundary points at $i = N-1, N$, where N is the number of grid points. We use the same method also for the η- and ζ-directions.

To suppress numerical instabilities that can arise from boundary conditions, unresolved scales, and mesh nonuniformities [6], we perform spatial filtering using the following tridiagonal filter in [7]:

$$\alpha_f \bar{f}_{i-1} + \bar{f}_i + \alpha_f \bar{f}_{i+1} = \sum_{n=0}^{3} \frac{a_n}{2}(f_{i+n} + f_{i-n}) \tag{6}$$

where the coefficients a_n are given in [7]. The parameter α_f must satisfy the inequality $-0.5 < \alpha_f < 0.5$. The seven-point stencil for computing the right-hand side is inapplicable for near-boundary points at $i = 1, 2, 3$ and $i = N-2, N-1, N$. For $i = 2, 3$, we use the following alternative formula:

$$\alpha_f \bar{f}_{i-1} + \bar{f}_i + \alpha_f \bar{f}_{i+1} = \sum_{n=1}^{7} a_{n,i} f_n \tag{7}$$

where the coefficients $a_{n,i}$ are also given in [7]. Similar formulations are applied for near-boundary points at $i = N-2, N-1$. Boundary points at $i = 1$ and $i = N$ are left unfiltered.

Algorithm 1 Spatial differentiation and filtering via transposition

1: Organize local data by post-transposition location
2: Use all-to-all communication to perform transposition
3: Set up right-hand sides locally
4: Solve tridiagonal systems locally
5: Organize computation results by pre-transposition location
6: Use all-to-all communication to restore original data layout

Both spatial differentiation and spatial filtering give rise to tridiagonal linear systems that are diagonally-dominant, except for the first and last rows in the case of spatial differentiation.

B. Transposition Scheme

We utilized a transposition scheme [1] in our previous efforts in implementing the above numerical methods.

In the transposition scheme, we partition the computational space along the ζ-direction and assign each partition to a processor. We use four-dimensional arrays to store values of flow variables in each grid partition, using three dimensions for curvilinear coordinates and one dimension for variable selection. This simple data layout enables compact representation of the flow field and allows for basic data aggregation in communication. Under this data layout, pointwise computation and spatial differentiation and filtering along the ξ- and η-directions are straightforward because they are local to each processor.

Spatial differentiation and filtering along the ζ-direction are global operations for they require values at all points on each grid line along the ζ-direction, which are scattered across all processors. To satisfy this requirement, we transpose the computational space across all processors as Fig. 1 illustrates so that it becomes partitioned along the η-direction. After that, tridiagonal systems for ζ-derivatives and filtering become local to the processors and can be solved with perfect parallelism. Finally, we reverse transpose the computational space to restore its original configuration. Algorithm 1 offers a synopsis of the steps of spatial differentiation and filtering. Within each time step, pointwise computation and spatial differentiation alternate several times and are followed by a spatial filtering at the end.

We use LAPACK for solving the tridiagonal systems arising from spatial differentiation and filtering, which gives the scheme high accuracy as well as high intranode efficiency. However, the scheme also has a major drawback that every spatial differentiation and filtering along the ζ-direction involves two all-to-all communications for transposing the computational space. In each time step, ζ-derivatives of as many as 45 flow variables and intermediate values are computed, and spatial filtering is performed on five flow variables along the ζ-direction. The huge amount of network traffic and network congestion due to the complex internode communication pattern induced by transposition necessarily hampers the efficiency of our large eddy simulations. In a recent code portability test after we moved to Kraken, a Cray XT5 cluster available from TeraGrid, as our primary computing platform, we measured that the communication cost was three times as high as what we had seen on Big Ben, a slower Cray XT3 cluster also available from TeraGrid. Such an effect will become even more prominent when we move to computing platforms with faster processor speed relative to interconnection network bandwidth.

III. Incorporating a SPIKE-based Tridiagonal Linear System Solver

To reduce the communication overhead, we consider replacing transposition with a new parallel tridiagonal solver based on the truncated SPIKE algorithm [3]. The SPIKE algorithm is a parallel hybrid solver for narrow-banded linear systems. For the purpose of our large eddy simulations, we apply it with specialization for tridiagonal linear systems. We note that discussion of the same algorithm had appeared in [8], but we follow the formulation in [3].

A. Basic Algorithm

For the sake of clarity in presentation, we assume that four processors are utilized. Generalization is straightforward.

To solve a tridiagonal linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{f}$, the SPIKE algorithm partitions $\boldsymbol{A}$ into four block rows:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 & b_1 & & \\ c_2 & \boldsymbol{A}_2 & b_2 & \\ & c_3 & \boldsymbol{A}_3 & b_3 \\ & & c_4 & \boldsymbol{A}_4 \end{bmatrix} \tag{8}$$

with tridiagonal blocks $\boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3, \boldsymbol{A}_4$ and off-diagonal elements b_1, b_2, b_3 and c_2, c_3, c_4. The block row containing $\boldsymbol{A}_k$ is assigned to processor k. The algorithm extracts the diagonal blocks from $\boldsymbol{A}$ to form a block diagonal matrix $\boldsymbol{D} = \mathrm{diag}(\boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3, \boldsymbol{A}_4)$ and computes the factorization

$$\boldsymbol{A} = \boldsymbol{D}\boldsymbol{S}. \tag{9}$$

Solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{f}$ then becomes equivalent to solving $\boldsymbol{S}\boldsymbol{x} = \boldsymbol{g}$ with $\boldsymbol{g} = \boldsymbol{D}^{-1}\boldsymbol{f}$. Computation of the matrix $\boldsymbol{S} = \boldsymbol{D}^{-1}\boldsymbol{A}$ can be done with perfect parallelism.

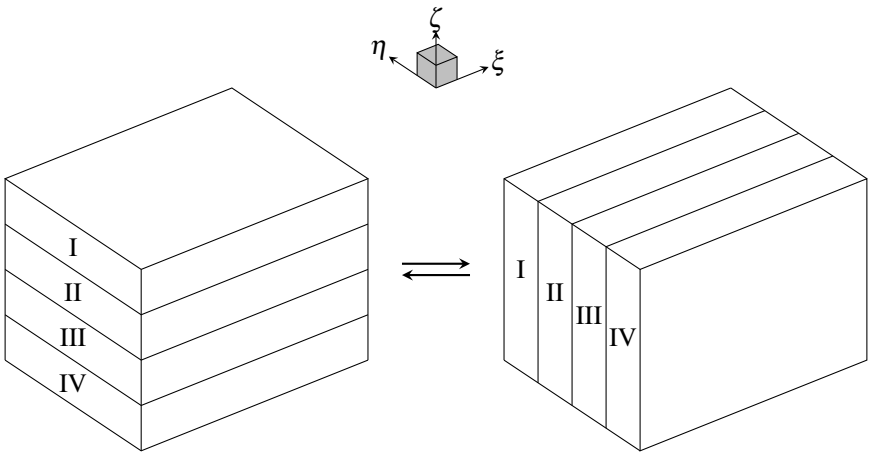


Figure 1. Transposition of the computational space

The matrix $\boldsymbol{S}$ is called the *spike matrix* for having the form

$$\boldsymbol{S} = \begin{bmatrix} \boxed{\begin{matrix} 1 & & * \\ & \ddots & \vdots \\ & & 1 \end{matrix}} & \begin{matrix} * \\ \vdots \\ * \end{matrix} \boldsymbol{v}_1 & & \\ \begin{matrix} * & \cdots & * \end{matrix} & \boxed{\begin{matrix} 1 & & * \\ & \ddots & \vdots \\ & & 1 \end{matrix}} & \begin{matrix} * \\ \vdots \\ * \end{matrix} \boldsymbol{v}_2 & \\ \boldsymbol{w}_2 & \begin{matrix} * & \cdots & * \end{matrix} & \boxed{\begin{matrix} 1 & & * \\ & \ddots & \vdots \\ & & 1 \end{matrix}} & \begin{matrix} * \\ \vdots \\ * \end{matrix} \boldsymbol{v}_3 \\ & \boldsymbol{w}_3 & \begin{matrix} * & \cdots & * \end{matrix} & \boxed{\begin{matrix} 1 & & * \\ & \ddots & \vdots \\ & & 1 \end{matrix}} \\ & & \boldsymbol{w}_4 & \begin{matrix} * & \cdots & * \end{matrix} \end{bmatrix} \tag{10}$$

with the "spikes"

$$\boldsymbol{v}_k = \boldsymbol{A}_k^{-1} \begin{bmatrix} \boldsymbol{0} \\ b_k \end{bmatrix}, \tag{11a}$$

$$\boldsymbol{w}_k = \boldsymbol{A}_k^{-1} \begin{bmatrix} c_k \\ \boldsymbol{0} \end{bmatrix}. \tag{11b}$$

Observe that in the system $\boldsymbol{S}\boldsymbol{x} = \boldsymbol{g}$, or

$$\begin{bmatrix} 1 & \vdots & & & & & \\ \ddots & \boxed{\begin{matrix} 1 & v_1^{(b)} \\ w_2^{(t)} & 1 \end{matrix}} & & & & & \\ & \vdots & \ddots & \boxed{\begin{matrix} 1 & v_2^{(b)} \\ w_3^{(t)} & 1 \end{matrix}} & & & \\ & & \vdots & \ddots & \boxed{\begin{matrix} 1 & v_3^{(b)} \\ w_4^{(t)} & 1 \end{matrix}} & & \\ & & & \vdots & \ddots & \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} \vdots \\ \boxed{\begin{matrix} x_1^{(b)} \\ x_2^{(t)} \end{matrix}} \\ \vdots \\ \boxed{\begin{matrix} x_2^{(b)} \\ x_3^{(t)} \end{matrix}} \\ \vdots \\ \boxed{\begin{matrix} x_3^{(b)} \\ x_4^{(t)} \end{matrix}} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \boxed{\begin{matrix} g_1^{(b)} \\ g_2^{(t)} \end{matrix}} \\ \vdots \\ \boxed{\begin{matrix} g_2^{(b)} \\ g_3^{(t)} \end{matrix}} \\ \vdots \\ \boxed{\begin{matrix} g_3^{(b)} \\ g_4^{(t)} \end{matrix}} \\ \vdots \end{bmatrix}, \tag{12}$$

when written in full, as marked by dashed rectangles, a block tridiagonal system

$$\begin{bmatrix} \hat{\boldsymbol{T}}_1 & \hat{\boldsymbol{V}}_1 & \\ \hat{\boldsymbol{W}}_2 & \hat{\boldsymbol{T}}_2 & \hat{\boldsymbol{V}}_2 \\ & \hat{\boldsymbol{W}}_3 & \hat{\boldsymbol{T}}_3 \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{x}}_1 \\ \hat{\boldsymbol{x}}_2 \\ \hat{\boldsymbol{x}}_3 \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{g}}_1 \\ \hat{\boldsymbol{g}}_2 \\ \hat{\boldsymbol{g}}_3 \end{bmatrix} \tag{13}$$

where

$$\hat{\boldsymbol{T}}_k = \begin{bmatrix} 1 & v_k^{(b)} \\ w_{k+1}^{(t)} & 1 \end{bmatrix}, \tag{14a}$$

$$\hat{\boldsymbol{V}}_k = \begin{bmatrix} 0 & 0 \\ 0 & v_{k+1}^{(t)} \end{bmatrix}, \tag{14b}$$

$$\hat{\boldsymbol{W}}_k = \begin{bmatrix} w_k^{(b)} & 0 \\ 0 & 0 \end{bmatrix}, \tag{14c}$$

$$\hat{\boldsymbol{x}}_k = \begin{bmatrix} x_k^{(b)} \\ x_{k+1}^{(t)} \end{bmatrix}, \tag{14d}$$

$$\hat{\boldsymbol{g}}_k = \begin{bmatrix} g_k^{(b)} \\ g_{k+1}^{(t)} \end{bmatrix}, \tag{14e}$$

can be extracted from (12) independently of other rows and columns as shown in (13). We call this system the *reduced system* and denote it by $\hat{\boldsymbol{S}}\hat{\boldsymbol{x}} = \hat{\boldsymbol{g}}$. The unknowns in $\hat{\boldsymbol{x}}$ found from solving the reduced system can be backsubstituted into (12) to obtain the complete solution $\boldsymbol{x}$ in parallel.

Multiple variants are possible for the SPIKE algorithm, depending on the method of solving the reduced system. For our large eddy simulations, we choose to implement the truncated SPIKE algorithm for its superior scalability over the other variants. The truncated SPIKE algorithm derives from the fact shown in [9] that for diagonally-dominant systems, magnitudes of elements on $\boldsymbol{v}_k$ and $\boldsymbol{w}_k$ decay exponentially as they lie further away from the diagonal. Hence, it is reasonable that we ignore the elements $v_k^{(t)}$ and $w_k^{(b)}$ at the spike tips, truncating the spikes $\boldsymbol{v}_k$ and $\boldsymbol{w}_k$. This reduces (13) to a

block diagonal system, which can be solved also with perfect parallelism.

To compensate for inaccuracies introduced by truncation of the spikes, the truncated SPIKE algorithm is typically wrapped inside an outer iterative scheme to improve the accuracy of the solution. For our large eddy simulations, we select iterative refinement as the outer iterative scheme to accompany the truncated SPIKE algorithm because alternative methods such as BiCGSTAB [10] and QMR [11] contain evaluations of dot products, which require global communication. Iterative refinement, in constrast, solely computes matrix-vector products, which, for tridiagonal systems, require communication between neighboring processors only.

To perform iterative refinement on the reduced system in (13), we start with the initial guess $\hat{\boldsymbol{x}}^{(0)} = \hat{\boldsymbol{T}}^{-1}\hat{\boldsymbol{g}}$ where

$$\hat{\boldsymbol{T}} = \mathrm{diag}(\hat{\boldsymbol{T}}_1, \hat{\boldsymbol{T}}_2, \hat{\boldsymbol{T}}_3) \tag{15}$$

and the corresponding residual $\hat{\boldsymbol{r}}^{(0)} = \hat{\boldsymbol{g}} - \hat{\boldsymbol{S}}\hat{\boldsymbol{x}}^{(0)}$, and update the iterate and residual via

$$\hat{\boldsymbol{x}}^{(n+1)} = \hat{\boldsymbol{x}}^{(n)} + \hat{\boldsymbol{T}}^{-1}\hat{\boldsymbol{r}}^{(n)}, \tag{16a}$$

$$\hat{\boldsymbol{r}}^{(n+1)} = \hat{\boldsymbol{r}}^{(n)} - \hat{\boldsymbol{S}}\hat{\boldsymbol{T}}^{-1}\hat{\boldsymbol{r}}^{(n)}. \tag{16b}$$

B. Limiting the Number of Iterations of Iterative Refinement

The primary motivation for replacing transposition with a SPIKE-based tridiagonal linear system solver is to reduce the number of bytes transmitted over the interconnection network. Inside the SPIKE-based solver, the main contributor of communication is solving the block tridiagonal reduced system. In solving the reduced system, the amount of communication is determined by the number of iterations of iterative refinement. We want to run sufficiently many iterations so that numerical accuracy is not sacrificed, while in the mean time, we also want to run as few iterations as possible so that communication is truly reduced.

In classical iterative methods for solving linear systems, the relative residual is monitored at runtime so that as soon as it drops below the tolerance level, the iteration can terminate. However, in the scenario of our large eddy simulations, keeping track of the relative residual at runtime will defeat the purpose of reducing communication because computation of the relative residual involves global communication across all processors. Therefore, we want to calculate in advance the minimum number of iterations that will guarantee a relative residual below the tolerance level in the worst case.

We calculate the minimum number of iterations needed as follows. Recall that in the SPIKE algorithm, solving the original system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{f}$, or effectively $\boldsymbol{S}\boldsymbol{x} = \boldsymbol{g}$, is accomplished via solving the reduced system in (13) and performing back-substitution. Ignoring $v_k^{(t)}$ and $w_k^{(b)}$ in the reduced system is equivalent to replacing $\boldsymbol{S}$ with $\tilde{\boldsymbol{S}}$, a copy of $\boldsymbol{S}$ with $v_k^{(t)}$ and $w_k^{(b)}$ ignored, in the system $\boldsymbol{S}\boldsymbol{x} = \boldsymbol{g}$. The iteration in (16) is equivalent to an alternative process of iterative refinement initialized with the initial guess $\boldsymbol{x}^{(0)} = \tilde{\boldsymbol{S}}^{-1}\boldsymbol{g}$ and the corresponding residual $\tilde{\boldsymbol{r}}^{(0)} = \boldsymbol{g} - \boldsymbol{S}\boldsymbol{x}^{(0)} = (\boldsymbol{I} - \boldsymbol{S}\tilde{\boldsymbol{S}}^{-1})\boldsymbol{g}$. Each iteration of iterative refinement updates the iterate and the residual via

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \tilde{\boldsymbol{S}}^{-1}\tilde{\boldsymbol{r}}^{(n)}, \tag{17a}$$

$$\tilde{\boldsymbol{r}}^{(n+1)} = \tilde{\boldsymbol{r}}^{(n)} - \boldsymbol{S}\tilde{\boldsymbol{S}}^{-1}\tilde{\boldsymbol{r}}^{(n)}. \tag{17b}$$

Hence, the residual after the nth iteration is

$$\tilde{\boldsymbol{r}}^{(n)} = (\boldsymbol{I} - \boldsymbol{S}\tilde{\boldsymbol{S}}^{-1})^{n+1}\boldsymbol{g}. \tag{18}$$

The residual with respect to the original system is thus

$$\begin{aligned} \boldsymbol{r}^{(n)} &= \boldsymbol{f} - \boldsymbol{A}\boldsymbol{x}^{(n)} \\ &= \boldsymbol{D}(\boldsymbol{g} - \boldsymbol{S}\boldsymbol{x}^{(n)}) \\ &= \boldsymbol{D}\tilde{\boldsymbol{r}}^{(n)} \\ &= \boldsymbol{D}(\boldsymbol{I} - \boldsymbol{S}\tilde{\boldsymbol{S}}^{-1})^{n+1}\boldsymbol{D}^{-1}\boldsymbol{f}. \end{aligned} \tag{19}$$

Therefore, the spectral norm relative residual $\|\boldsymbol{r}^{(n)}\|_2 / \|\boldsymbol{f}\|_2$ satisfies

$$\frac{\|\boldsymbol{r}^{(n)}\|_2}{\|\boldsymbol{f}\|_2} \leq \left\|\boldsymbol{D}(\boldsymbol{I} - \boldsymbol{S}\tilde{\boldsymbol{S}}^{-1})^{n+1}\boldsymbol{D}^{-1}\right\|_2 \tag{20}$$

where $\|\cdot\|_2$ denotes the spectral norm of a vector or a matrix. Given a tolerance level ε, we compute the smallest value of n such that $\left\|\boldsymbol{D}(\boldsymbol{I} - \boldsymbol{S}\tilde{\boldsymbol{S}}^{-1})^{n+1}\boldsymbol{D}^{-1}\right\|_2 < \varepsilon$ and take that as the minimum number of iterations needed. Since $\boldsymbol{A}$ is constant for any particular grid size, we perform the computation offline using MATLAB.

C. Overlapping Communication and Computation

In addition to limiting the number of iterations of iterative refinement, we also invest in taking advantage of asynchoronous communication in as many occasions as possible to enable overlapping of communication and computation.

In our large eddy simulations, there are two main sources of opportunities for overlapping communication and computation, namely, setting up the right-hand sides for spatial differentiation and filtering, and iterative refinement inside the SPIKE-based tridigonal linear system solver.

In spatial differentiation and filtering, the right-hand sides are computed using a five-point stencil and a seven-point stencil, respectively. Between processors handling neighboring block rows in the SPIKE algorithm, two or three planes of data have to be exchanged depending on the case. In other words, each processor that is not assigned the top or bottom row block has to exchange data with both of its neighboring processors. We implement computation of the right-hand sides in such a way that each processor will start computation as soon as it receives data from the neighbor above it and defer waiting for data from the neighbor below until computation cannot proceed due to lack of data. Correspondingly, each processor will initiate asynchronous send of data to the neighbor below

Algorithm 2 Setting up right-hand sides

1: Let p be the number of processors
2: Let k be the processor number
3: Post request to receive from processor $k-1$ if $k \neq 1$
4: Post request to receive from processor $k+1$ if $k \neq p$
5: Initiate send to processor $k+1$ if $k \neq p$
6: Initiate send to processor $k-1$ if $k \neq 1$
7: Wait for data from processor $k-1$ if $k \neq 1$
8: Compute all except the last two or three rows of the right-hand sides
9: Wait for data from processor $k+1$ if $k \neq p$
10: Compute the remaining rows of the right-hand sides

it before to the one above it so that computation on the destination processor can start as early as possible. The algorithmic steps are summarized in Algorithm 2.

Inside the SPIKE-based tridiagonal linear system solver, iterative refinement also involves exchange of data between neighboring processors almost identical to that in setting up the right-hand sides, only differing by the amount of communication—only one plane of data is exchanged between neighboring processors. In comparison with setting up the right-hand sides, asynchoronous communication exhibits greater necessity in this scenario because such data exchange occurs repeatedly in each iteration of iterative refinement.

In our implementation, each processor posts all its requests to receive data up front so that each receive operation can start as soon as its neighbors have the data to exchange ready; for sending data, each processor allocates a separate buffer for data from each iteration so that send operations of different iterations can execute concurrently. Algorithm 3 provides a more precise view of where communication primitives appear.

Algorithm 3 Iterative refinement on reduced system

1: Let p be the number of processors
2: Let k be the processor number
3: Perform a downward shift of the reduced system $\hat{\boldsymbol{S}}\hat{\boldsymbol{x}} = \hat{\boldsymbol{g}}$ by one row so that each of processors 2 to p holds a complete two-row block
4: **if** $k \neq 1$ **then**
5: Let τ be the offline-decided number of iterations
6: Post τ requests to receive from processor $k+1$ if $k \neq p$
7: Post τ requests to receive from processor $k-1$ if $k \neq 2$
8: Compute local part of $\hat{\boldsymbol{x}}^{(0)}$
9: **for** $i = 1, 2, \ldots, \tau$ **do**
10: Initiate send of $x_{k-1}^{(b)}$ to processor $k+1$ if $k \neq p$
11: Initiate send of $x_{k-1}^{(t)}$ to processor $k-1$ if $k \neq 2$
12: Wait for data from processor $k+1$ if $k \neq p$
13: Wait for data from processor $k-1$ if $k \neq 2$
14: Compute local part of $\hat{\boldsymbol{r}}^{(i-1)}$
15: Compute local part of $\hat{\boldsymbol{x}}^{(i)}$
16: **end for**
17: **end if**
18: Perform an upward shift of the reduced system by one row to restore the original data layout

IV. Experimental Results

A. Implementation

Both of our original and new LES implementations are written in Fortran 90 and use the Message Passing Interface (MPI) for internode communication.

In the original implementation, transposition of the computational space is accomplished via the `MPI_ALLTOALL` subroutine. The LAPACK subroutine pair `DGTTRF`/`DGTTRS` is used for solving the tridiagonal linear systems on each processor.

In the new implementation, we replace the portions of the program that rely on the `MPI_ALLTOALL` subroutine with three new modules that we have created to incorporate the SPIKE-based tridiagonal linear system solver. Two of the modules are responsible for setting up the right-hand sides of the spatial differentiation and filtering; the third module implements the solver proper. In all three modules, we implement flexible interfaces that accept three-dimensional and four-dimensional arrays, whose usage will be clear in subsection IV-B.

B. Experiment Setup

To compare our new LES implementation against the baseline of our original implementation based on the transposition scheme, we use a test problem on a $768 \times 768 \times 768$ grid totaling 453 million grid points. Although for a full LES the run typically has to last at least 100 000 time steps of Runge-Kutta integration, for benchmarking purposes, we limit each implementation to run ten time steps. We measure the running time spent on Runge-Kutta integration only, excluding the costs of startup and shutdown. We run each implementation three times and calculate the average time.

We test three different flavors of using the SPIKE-based tridiagonal linear system solver by feeding it with

- values at one plane of grid points perpendicular to the η-direction at a time, or,
- values at all grid points of one flow variable at a time, or,
- values at all grid points of all flow variables at a time.

They are all implemented using the flexible interfaces provided by the new modules. Feeding data in small chunks to the solver achieves better data locality at the expense of higher communication startup cost; at the opposite end, feeding data in one large batch saves on communication startup but risks inferior data locality. For future reference, we name the three flavors "SPIKE/plane", "SPIKE/var" and "SPIKE/all", respectively.

Table II
HARDWARE CONFIGURATIONS OF RANGER AND KRAKEN

	Ranger	Kraken
Cluster model	Sun Constellation	Cray XT5
Processor model	AMD "Barcelona"	AMD "Istanbul"
Architecture	x86-64	x86-64
Sockets per node	4	2
Cores per socket	4	6
Clock frequency	2.3 GHz	2.6 GHz
Memory per node	32 GB	16 GB

We do five separate sets of test runs using 24, 32, 48, 64 and 96 nodes, respectively, and utilize one processor on each node. These configurations correspond to block row sizes in the SPIKE algorithm of 32, 24, 16, 12 and 8 rows, respectively. We base our estimates for the number of iterations of iterative refinement in the SPIKE-based tridiagonal linear system solver on the cases with 16-row blocks, using the unit round-off of double-precision floating numbers as the tolerance level, i.e., $\varepsilon = 2^{-52} \approx 2.22 \times 10^{-16}$. Our calculations show that two iterations of iterative refinement are necessary for spatial differentiation, and seven iterations for spatial filtering. We use these two numbers for all block row sizes because they are sufficient for block sizes of 16 rows or larger and provide reasonable accuracy for that of 8 rows.

We carry out the performance experiments on Ranger and Kraken, two clusters available from TeraGrid. Hardware configurations of Ranger and Kraken are listed in Table II. Limited by memory capacity, we are unable to test the 24-node configuration on Kraken.

C. Results

We plot the speedup achieved by the new LES implementation with respect to the original implementation in Fig. 2, and the speedup achieved by both implementations with respect to the platform baselines (the 24-node case on Ranger and the 32-node case on Kraken) of the original implmentation in Fig. 3. It is evident from Fig. 2 that the new implementation attains significantly higher performance than the original implementation on both Ranger and Kraken. The geometric-mean speedup across the three different flavors of applying the SPIKE-based tridiagonal linear system solver is 223 % on Ranger and 155 % on Kraken. The figure being higher on Ranger than on Kraken, as also observable in both Fig. 2 and Fig. 3, results from the fact that the interconnection network of Ranger is slower than that of Kraken with respect to processor speed. This shows that the new implementation is more adaptable to limited interconnection network bandwidth. The differences between the three flavors of the new implementation are minor. We notice that as shown in Fig. 2, speedup achieved in the 64- and 96-node cases is slightly lower than in the other cases. We believe that this is likely due to imperfect physical embedding of the logical linear processor grid during the experiments, which induces higher communication overhead than theory.

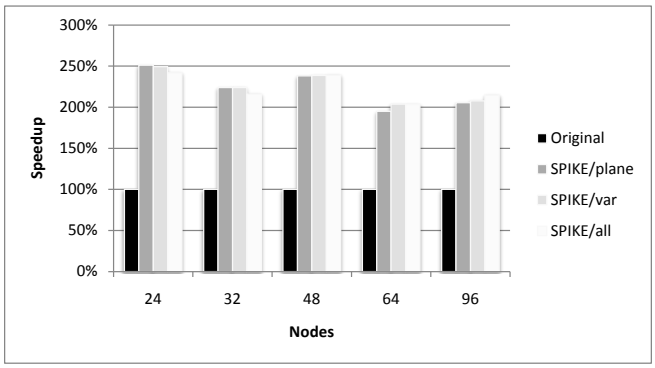
We plot the scalability of the new implementation as the speedup with respect to its own platform baseline in Fig. 4. As Fig. 4 shows, the new implementation achieves almost perfect scalability on both Ranger and Kraken, especially in the "SPIKE/all" flavor. The geometric-mean parallel efficiency of the new implementation across all three flavors is 95 % on Ranger and 98 % on Kraken, which indicates very good scalability. (The parallel efficiency is defined as $\frac{p_{\text{base}} T_{\text{base}}}{pT}$ where p and T are the number of nodes and running time, respectively, and p_{base} and T_{base} are those of the platform baseline.)

We use the Cray Performance Analysis Tools (CrayPat) [12] available on Kraken to instrument the compiled binaries to measure the communication overhead. We profile the two implementations over five time steps instead of ten to shorten the profiling time. Fig. 5 shows the communication overhead, which includes time spent in both MPI subroutines and the necessary user code that arranges data layout for MPI subroutines, in seconds and as the percentage of total time. Thanks to the riddance of calls to the `MPI_ALLTOALL` subroutine, the new implementation costs significantly less in communication compared to the original implementation in terms of both the absolute time and the percentage of total time except for the "SPIKE/plane" flavor in the 64-node and 96-node cases. Fig. 6, which shows only the time spent in MPI subroutines, reveals the reason for such exception. Despite having the same amount of network traffic as the other two flavors, the "SPIKE/plane" flavor has very high communication startup overhead for making exccessively many calls to MPI subroutines (up to 1.7 million per processor) due to its plane-by-plane nature.

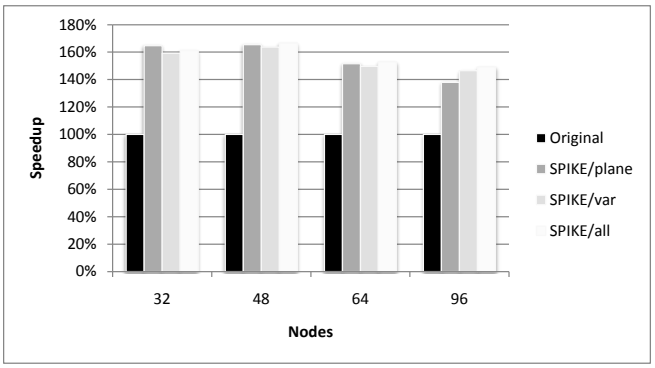
We are unable to carry out the same experiments on Ranger because Ranger does not offer a binary instrumentor, and the source code-based instrumentor does not correctly handle our Fortran 90 source code.

V. CONCLUSIONS AND FUTURE WORK

In this study, we investigate the feasibility of reducing communication overhead in large eddy simulations by employing a new parallel tridiagonal linear system solver based on the truncated SPIKE algorithm to replace the transposition scheme. Experimental performance data show that application of the new solver delivers much better performance over the transposition scheme and significantly reduces the communication overhead. These results establish the foundation of our future work, in which we will extend the idea of applying the new solver by partitioning the computational space in all three directions. This will allow us to harness parallelism in three dimensions and take advantage of the computing power of a large number of processors to accelerate the simulation process.

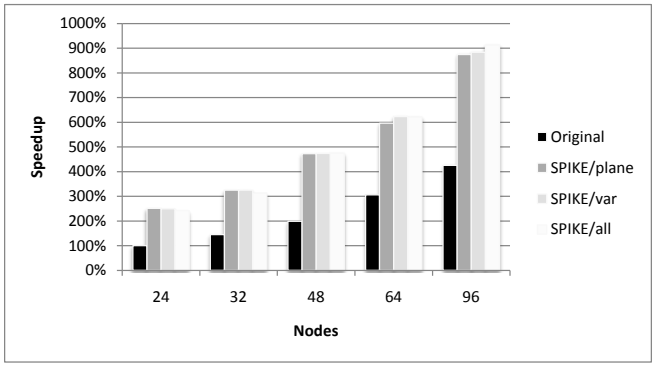


(a) Ranger

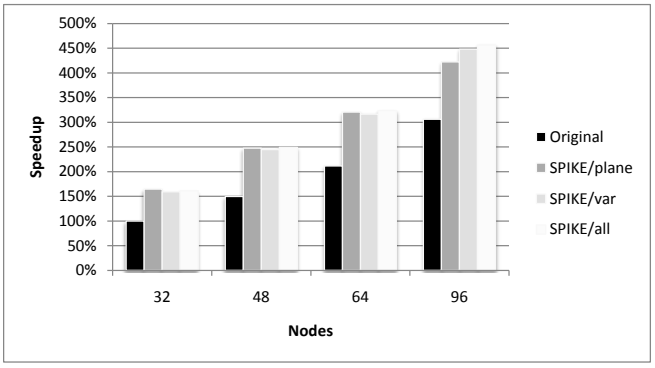


(b) Kraken

Figure 2. Speedup with respect to the orignal implementation

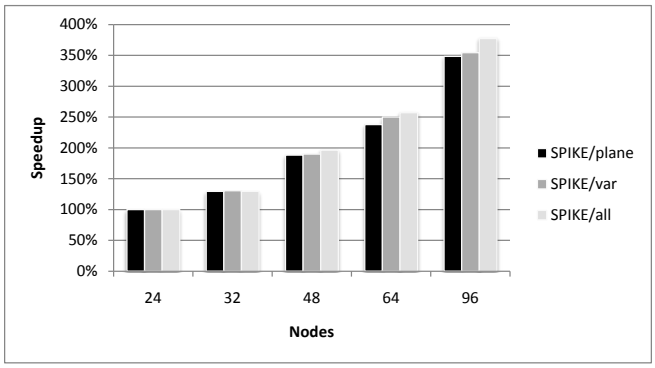


(a) Ranger

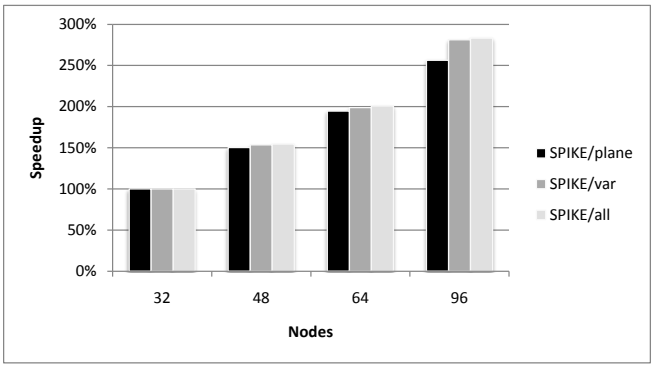


(b) Kraken

Figure 3. Speedup with respect to platform baseline of the original implementation

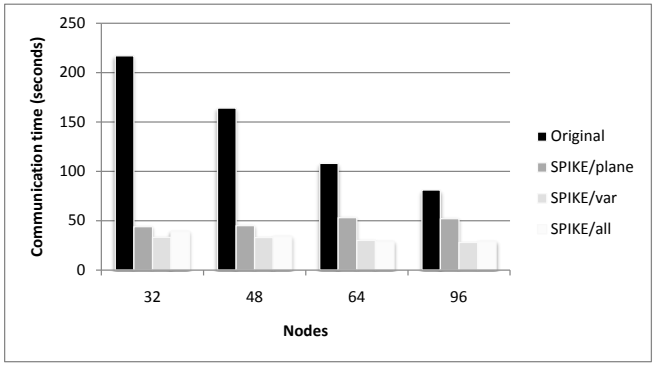


(a) Ranger

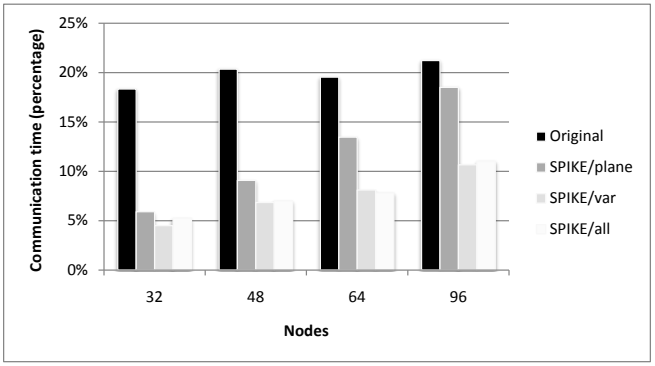


(b) Kraken

Figure 4. Speedup with respect to platform baseline

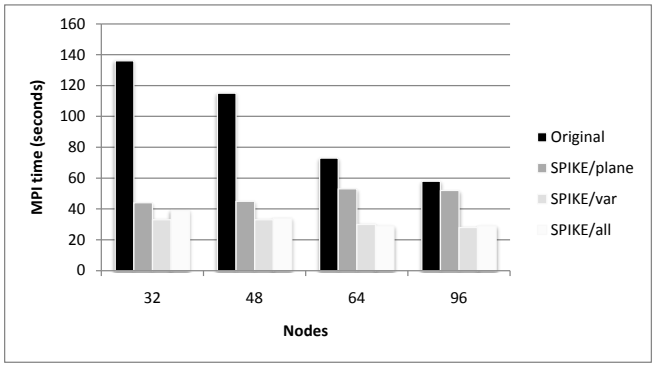


(a) Communication time in seconds

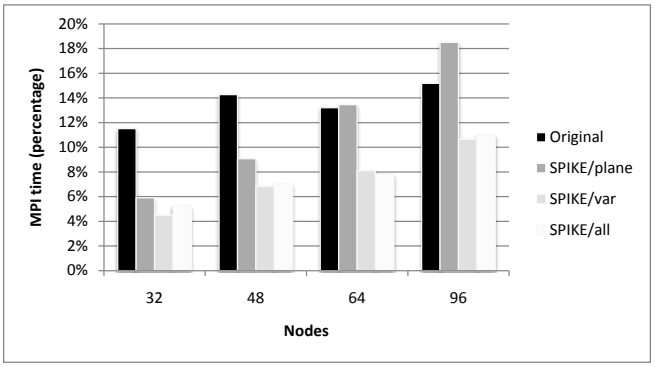


(b) Communication time as percentage of total time

Figure 5. Communication overhead on Kraken



(a) MPI time in seconds



(b) MPI time as percentage of total time

Figure 6. MPI time on Kraken

ACKNOWLEDGMENT

Our thanks go to Ali Uzun, whose LES code base has greatly reduced our workload in creating the new LES implementation that is benchmarked in this study.

This research is primarily sponsored by the National Science Foundation (NSF) via Award OCI-0904675 under its PetaApps program. This research is further supported by the NSF via Award CCF-0811587 and through TeraGrid [13] resources provided by TACC and NICS as allocations TG-ASC090008 and TG-ASC040044N. TeraGrid systems are hosted by Indiana University, LONI, NCAR, NCSA, NICS, ORNL, PSC, Purdue University, SDSC, TACC, and UC/ANL. The second author is also supported by a Google fellowship.

REFERENCES

[1] A. Uzun, "3-D large eddy simulation for jet aeroacoustics," Ph.D. dissertation, School of Aeronautics and Astronautics, Purdue University, 2003.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: a portable linear algebra library for high-performance computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 2–11.

[3] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: the SPIKE algorithm," *Parallel Computing*, vol. 32, no. 2, pp. 177–194, 2006, Parallel Matrix Algorithms and Applications (PMAA'04).

[4] A. Uzun, G. A. Blaisdell, and A. S. Lyrintzis, "Application of compact schemes to large eddy simulation of turbulent jets," *Journal of Scientific Computing*, vol. 21, no. 3, pp. 283–319, 2004.

[5] S. K. Lele, "Compact finite difference schemes with spectral-like resolution," *Journal of Computational Physics*, vol. 103, no. 1, pp. 16–42, 1992.

[6] E. K. Koutsavdis, G. A. Blaisdell, and A. S. Lyrintzis, "Compact schemes with spatial filtering in computational aeroacoustics," *AIAA Journal*, vol. 38, pp. 713–715, 2000.

[7] M. R. Visbal and D. V. Gaitonde, "Very high-order spatially implicit schemes for computational acoustics on curvilinear meshes." *Journal of Computational Acoustics*, vol. 9, no. 4, pp. 1259–1286, 2001.

[8] J. J. Dongarra and A. H. Sameh, "On some parallel banded system solvers," *Parallel Computing*, vol. 1, no. 3–4, pp. 223–235, 1984.

[9] C. C. K. Mikkelsen and M. Manguoglu, "Analysis of the truncated SPIKE algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 4, pp. 1500–1519, 2008.

[10] H. A. van der Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[11] R. W. Freund and N. M. Nachtigal, "QMR: a quasi-minimal residual method for non-Hermitian linear systems," *Numerische Mathematik*, vol. 60, no. 1, pp. 315–339, 1991.

[12] Cray Inc., *Using Cray Performance Analysis Tools*. Cray Inc., 2009, S-2376-50.

[13] C. Catlett et al., "TeraGrid: Analysis of organization, system architecture, and middleware enabling new types of applications," in *High Performance Computing and Grids in Action*, ser. Advances in Parallel Computing, L. Grandinetti, Ed. Amsterdam: IOS Press, 2007.