computation

MDPI

Article

Interface Splitting Algorithm: A Parallel Solution to Diagonally Dominant Tridiagonal Systems

Arpiruk Hokpunna

Department of Mechanical Engineering, Faculty of Engineering, Chiang Mai University, Chiang Mai 50200, Thailand; arpiruk.hok@eng.cmu.ac.th

Abstract: We present an interface-splitting algorithm (ITS) for solving diagonally dominant tridiagonal systems in parallel. The construction of the ITS algorithm profits from bidirectional links in modern networks, and it only needs one synchronization step to solve the system. The algorithm trades some necessary accuracy for better parallel performance. The accuracy and the performance of the ITS algorithm are evaluated on four different parallel machines of up to 2048 processors. The proposed algorithm scales very well, and it is significantly faster than the algorithm used in ScaLAPACK. The applicability of the algorithm is demonstrated in the three-dimensional simulations of turbulent channel flow at Reynolds number 41,430.

Keywords: parallel numerical algorithm; tridiagonal equations; scalable computing; compact scheme; Navier–Stokes equations; high order method

check for updates

Citation: Hokpunna, A. Interface Splitting Algorithm: A Parallel Solution to Diagonally Dominant Tridiagonal Systems. *Computation* **2023**, *11*, 187. https://doi.org/10.3390/computation11090187

Academic Editor: Francesco Cauteruccio

Received: 12 July 2023
Revised: 18 August 2023
Accepted: 1 September 2023
Published: 21 September 2023

1. Introduction

The numerical solution process for tridiagonal systems is a crucial task in scientific computing, such as in, for example, wavelets [1–3], spline interpolations [2], and numerical simulations of partial differential equations, such as heat transfer problems [4], convection–diffusion phenomena [5,6], and computational fluid dynamics [7,8], among others. Traditionally, the approximation schemes used in the finite difference method (FDM) and the finite volume method (FVM) are explicit schemes that are very easy to parallelize by using ghost cells that are padded to the actual computing domain. These ghost cells act as a surrogate for the true data, which are actually being solved on other processors. However, explicit schemes often exhibit relatively poor resolving power compared to implicit schemes. Lele [5] proposed to use compact schemes to approximate the first and second derivatives. He shows that, in a broad spectrum problem where the fundamental physics contains a vastly different range of length scale, a compact scheme is superior to the explicit scheme of the same order of accuracy. For example, instead of computing the second derivative through an explicit scheme, he suggests using compact higher-order schemes such as

$$l_i T''_{i-1} + d_i T''_i + r_i T''_{i+1} = \kappa_i \phi_{i+1} + \varsigma_i \phi_{i-1}, \quad (1)$$

which is considerably more efficient than the explicit scheme of the same convergence rate. The parameters l_i, d_i, r_i, κ_i, and ς_i are the approximation coefficients. These compact schemes have far-reaching implications, serving as a foundation for diverse algorithms in various fields. This includes applications in Navier–Stokes equations [9–11], magnetohydrodynamics [12], acoustics [13,14], chemical reacting flow [15], as well as emerging discretization approaches like finite surface discretization [8,16,17] and the multi-moment method [6].

When numerical grids sufficiently capture the underlying phenomenon, employing high-order numerical methods can yield remarkable efficiency. Hokpunna et al. [7] demonstrate that a compact fourth-order scheme outperforms the classical second-order finite volume method (FVM) by a factor of 10. The efficiency improvement continues with the

sixth-order compact finite surface method [8], which is 2.7 times more effective than the compact fourth-order FVM. Thus, the integration of a compact scheme in physical simulations significantly accelerates scientific research. However, the equation mentioned above needs an efficient solution method for solving diagonally dominant tridiagonal systems in parallel. The implicit nature that enhances the approximation's resolving power introduces a directional dependency (in this case, i). The simplest problem configuration is illustrated in Figure 1. Efficient parallelization of the solution process in this context is not straightforward.

Applications in the previously mentioned areas often necessitate transient solutions with an extensive number of grid points. This scale is still challenging today even for massively parallel computer systems. For instance, isotropic turbulence simulations conducted on the Earth Simulator to explore turbulent flow physics involved up to 68 billion grid points [18–20]. Applying a compact scheme to this case involves solving tridiagonal systems with a system size of 4096 and a substantial number of right-hand sides (rhs), amounting to 4096^2. Similarly, in wall-bounded flows, Lee and Moser [21] utilized 121 billion grid points for solving one problem. The computational power required for these problems far surpasses the capabilities of shared memory computers and requires distributed memory computers for the simulation. Despite the remarkable scale of these simulations, the Reynolds numbers in these cases are still far smaller than those seen in industrial applications. Therefore, efficient parallel algorithms for solving such systems are in demand.

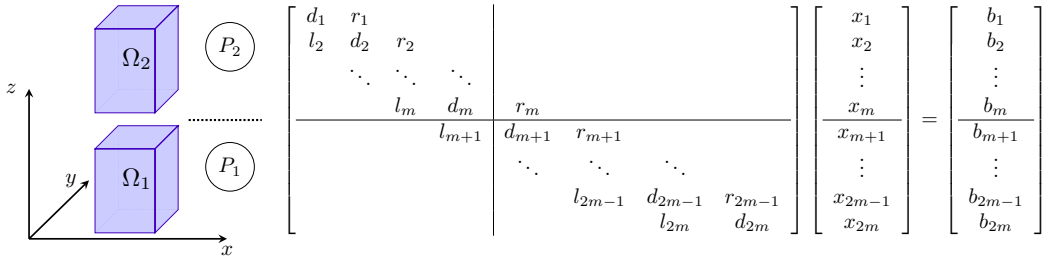


Figure 1. Decomposition of a domain $\Omega = \Omega_1 \cup \Omega_2$ on a 3D Cartesian grid system (left). Each subdomain is assigned to the corresponding processor, P_1 and P_2. The tridiagonal system $Ax = b$ of the compact approximation in z-direction (right) is split.

Let us consider a cooling down process of a metal slab that was initially heated with the temperature distribution $T(x, y, z, 0) = \Theta(x, y, z)$ i.e., $\forall (x, y, z) \in \Omega$, with the following governing equation:

$$\frac{\partial T}{\partial t} - \Gamma \Delta T = 0, \tag{2}$$

where Γ is the Péclet number, and the boundary condition is homogeneous. We can evolve the temperature in time by solving

$$T^{n+1} = T^n + \Gamma \int_{t_n}^{t_{n+1}} \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) dt. \tag{3}$$

The spatial approximation from the compact scheme (Equation (1)) can be used to obtain the three second derivatives in this equation. The implicitness of the compact scheme is only carried in one direction. Thus, for the above setting, we have three independent linear systems. If an explicit time integration is used for Equation (3), each second derivative of the Laplacian must be evaluated and then summed up to compute the time integral. The linear system of equation (LSE) at each time step is

$$\mathbf{AX} = \mathbf{B}, \tag{4}$$

where $\mathbf{X} = \left[\mathbf{x}_1 \mid \ldots \mid \mathbf{x}_{Nx\times Ny}\right]$ and $\mathbf{B} = \left[\mathbf{b}_1 \mid \ldots \mid \mathbf{b}_{Nx\times Ny}\right]$, the augmentation of the unknown and the right-hand vectors, respectively. Each unknown vector $\mathbf{x}_i$ is the second derivative containing Nz members.

Let us consider a simple setting depicted in Figure 1. Here, the domain is divided and distributed to two processors, P_1 and P_2. The two domains are separated at the midplane in the k direction. It is not straightforward to find the solution to the LSE of $\partial^2 T/\partial z^2$ because the right-hand sides are located on different processors. In this setting, the linear system for $\partial^2 T/\partial x^2$ and $\partial^2 T/\partial y^2$ can be solved without any problem because the needed data are located within the processor. It should be clear that, in a large scaled simulation, there will be multiple cuts in all three directions. The missing dependency across the interfaces has to be fulfilled by some means.

It is clear that the approximation in each tuple (i, j) normal to the xy-plane is independent of one another, and one can transfer half of the missing data to one processor, solve the systems, and then transfer the solution back to it. This approach is called the *transpose algorithm* [22]. This algorithm achieves the minimum number of floating point operations, but it has to relocate a lot of data. The minimum data transfer of this algorithm is $NxNyNz$. This approach is therefore not suitable for massively parallel computers in which the floating point operations are much faster than the intercommunication. Another approach that achieves the minimum number of floating point operations and also the minimum amount of data transfer is the *pipelined algorithm* [23]. In this approach, the tuples are solved successively after the dependency is satisfied. For example, the processor p_1 can start the forward elimination first on the $(1, 1)$ tuple until the mth row and then send the result to p_2. Once p_2 obtains the data, it can start the forward elimination on that tuple, and p_1 can work on the forward elimination of the next tuple. The simplest pipelined algorithm will send a real number $2NxNy$ times per interface. The performance of this algorithm depends heavily on the latency and availability of the communication system.

When designing an algorithm, the designer should be aware of the disproportion between the data transfer and computing power. Take the AMD 7763 processor as an example: it has a memory bandwidth (BW_{mem}) of 205 GBytes/s, while its theoretical peak performance reaches 3580 giga floating point operations per second (GFLOPS). This performance is based on the fused–multiplied–add operation that needs four data for each calculation. For the processor to work at the peak rate, it would need to obtain the data at 114,560 GB/s (for the double precision data). Let us call this the transfer rate BW_{peak}. The ratio BW_{peak}/BW_{mem} is about 560. This disparity requires algorithms to excel in terms of communication efficiency. In parallel computing, things become even more difficult regarding the interconnection between computing nodes. A dual-socket computing node with this setting linked to other nodes by a network bandwidth BW_{net} = 80 GB/s would have the ratio BW_{peak}/BW_{mem} = 2200. For a GPU cluster with a recent NVIDIA H100, the ratio of the BW_{peak}/BW_{net} is even higher at 16,320. And this unbalance keeps growing. This disparity between the computational power and the ability to communicate between computing units renders fine-grain parallel algorithms unsuitable for the current massively parallel computer systems. Coarse-grained parallelism such as that in [24–29] is thus preferable. These coarse-grained algorithms send data less frequently but in a larger package. Lawrie and Sameh [30], Bondeli [26], and Sun [27] developed algorithms specialized for tridiagonal systems with the diagonal dominant property. The *reduced parallel diagonal dominant algorithm* (RPDD) in [27] can use p processors to solve a tridiagonal system of n equations with γ right-hand sides using a complexity of $(5n/p + 4J + 1)\gamma$ operations (with some small number J described later). This algorithm is very efficient for this problem.

In this paper, we present a new interface-splitting algorithm (ITS) for solving diagonally dominant tridiagonal linear systems with the system size of n and γ right-hand sides on p processor. The algorithm has a complexity of $(5n/p + 4J - 4)\gamma$. The idea is to reduce the communications, truncate the data dependency, and fully take advantage of bi-directional communication. The proposed method makes use of exponential decay in

the inverse of diagonally dominant matrices explained in [31]. The proposed scheme is competitive and applicable for non-Toeplitz as well as periodic systems and non-uniform meshes. It has less complexity than the algorithm presented in [27], requires one synchronization step fewer, and the cost of the data transfer is potentially halved. Therefore, the proposed algorithm is less sensitive to load balancing and network congestion problems.

The presentation of this paper is organized as follows. First, we present the interface-splitting algorithm and then discuss its complexity and accuracy. The accuracy and the performance of the algorithm are evaluated on four specialized systems, starting with a single-node Intel Xeon 8168, and then SGI ALTIX 4700, IBM BladeCenter HS21XM, and NEC HPC 144Rb-1. The performance of the proposed algorithm is compared with the ScaLAPACK and the RPDD algorithms. After that, the application to Navier–Stokes equations is presented, followed by the conclusion and remarks.

2. Interface-Splitting Algorithm

2.1. Concept

The decomposition of the domain on different processors led to the problem mentioned earlier in Figure 1. The second processor cannot start the forward elimination because the interface value (x_m) is not known. However, if they have already computed the central row of $\mathbf{C} = \mathbf{A}^{-1}$, then they can both compute

$$x_m = \sum_{j=1}^{2m} c_{mj} b_j = \underbrace{\sum_{j=1}^{m} c_{mj} b_j}_{\text{from processor 1}} + \underbrace{\sum_{j=m+1}^{2m} c_{mj} b_j}_{\text{from processor 2}}$$

after exchanging a single real number. In this way, both processors can continue to solve their systems without further communication. It is well known that the inverse of the diagonally dominant tridiagonal matrix decays away from the diagonal, both row and column. The minimum decay rate of non-symmetric matrices has been developed in [32] and [31]. Thus, the interface value can be computed from the row inverse of a smaller matrix, e.g.,

$$\mathcal{A} = \left[\begin{array}{cc|cc} d_{m-1} & r_{m-1} & & \\ l_m & d_m & r_m & \\ ine & l_{m+1} & d_{m+1} & r_{m+1} \\ & & l_{m+2} & d_{m+2} \end{array}\right], \tag{5}$$

by computing

$$x_m \approx \sum_{j=m-1}^{m+2} (\mathcal{A}^{-1})_{mj} b_j = \underbrace{\sum_{j=m-1}^{m} (\mathcal{A}^{-1})_{mj} b_j}_{\text{from processor 1}} + \underbrace{\sum_{j=m+1}^{m+2} (\mathcal{A}^{-1})_{mj} b_j}_{\text{from processor 2}}.$$

This truncation introduces an error into the solution. However, in the system that originated from a simulation-based problem, there is a certain level of accuracy in how that particular system represents the true value. For example, the compact scheme in Equation (1) differs from the true second derivative by the size of the local truncation error (LTE). Any errors that are much smaller than LTE will not be observable. For example, in most approximations of the compact scheme above, the LTE should be larger than 1×10^{-10}. When such system is solved by a direct method under double precision, the error due to the solution process should be on the order of 1×10^{-14}. Using quadruple precision can reduce the solution error to about 1×10^{-32}, and the error of the differentiation would still be 1×10^{-10}.

In the next step, we explain the algorithm in detail, including the strategy for reducing the number of operations and controlling the truncation errors.

2.2. The Parallel Interface-Splitting Algorithm

For simplicity of notation, let us consider a tridiagonal system of size n with a single right-hand side,

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{6}$$

where $\mathbf{A}$ is a strictly diagonal dominant matrix, $\mathbf{A} = [l_i, d_i, r_i]$, $|d_i| > |l_i| + |r_i|$, and $l_1 = r_n = 0$. In order to solve this system in parallel, one can decompose the matrix using $\mathbf{A} = \mathbf{TQ}$ and solve the original system in two steps:

$$\mathbf{T}\mathbf{v} = \mathbf{b}, \tag{7}$$

$$\mathbf{Q}\mathbf{x} = \mathbf{v}. \tag{8}$$

The first equation computes the solution at the interface using the coefficients stored locally. This factorization can be considered as a preprocessing scheme where the right-hand side is modified such that the solution of the simpler block matrices delivers the desired result. The partitioning algorithm [24] and the parallel line solver [33] belong to this type.

Suppose that $\mathbf{A}$ is a tridiagonal matrix of size $n = pm$, where p is the number of processors and m is the size of our subsystems. The kth processor is holding the right-hand side $\mathbf{b}^k = b_j$, $(k-1)m + 1 \leq j \leq km$. Here, it is sufficient to consider one right-hand side. The application to multiple right-hand sides is straightforward.

Let $\mathbf{D}^k$ be the kth block subdiagonal matrix of $\mathbf{A}$, that is, $\mathbf{D}^k = \{\, a_{ij} \mid (k-1)m + 1 \leq i, j \leq km\}$ and $\mathbf{N}^k$ is the matrix $\mathbf{D}^k$, except the last row is replaced by that of the identity matrix:

$$\mathbf{N}^k = \left[\begin{array}{ccccc|c} d_{\nu+1} & r_{\nu+1} & & & & \\ l_{\nu+2} & d_{\nu+2} & r_{\nu+2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & l_{\nu+m-1} & d_{\nu+m-1} & & r_{\nu+m-1} \\ ine & & & & & 1 \end{array}\right] \tag{9}$$

Instead of using the block diagonal matrix of $\mathbf{D}^k$ as an independent subsystem as in the alternative algorithms, the interface-splitting algorithm builds the matrix $\mathbf{Q}$ up form the special block matrix $\mathbf{N}^k$:

$$\mathbf{Q} = \left[\begin{array}{cccc} \mathbf{N}^1 & & & \\ & \mathbf{N}^2 & & \\ & & \ddots & \\ & & & \mathbf{N}^p \end{array}\right] \tag{10}$$

The neighboring subsystems (kth and $(k+1)$th) are separated by the kth interface, which is the last row of the kth subsystem. Note that if we use $\mathbf{D}^k$ as an independent subsystem, the algorithm will be the transposed version of the PDD algorithm.

The algorithm corresponding to the selected decomposition is given by

$$\mathbf{A}^T \mathbf{z}_k = \widehat{\mathbf{g}}_{km}, \tag{11}$$

$$\mathbf{f}^k = \left(\mathbf{z}^k \cdot \mathbf{b}\right)\widehat{\mathbf{g}}_{km} + l_1^k\left(\mathbf{z}^{k-1} \cdot \mathbf{b}\right)\widehat{\mathbf{g}}_{(k-1)m+1}, \tag{12}$$

$$\mathbf{v}^k = \mathbf{b}^k - \mathbf{f}^k, \tag{13}$$

$$\mathbf{N}^k \mathbf{x}^k = \mathbf{v}^k, \tag{14}$$

where $\widehat{\mathbf{g}}_{km}$ is the vector whose kmth component is one and zero otherwise. The vector $\mathbf{z}^k$ is given by $z_j^k = c_{km,j}$, $1 \leq j \leq n$, with $\mathbf{C} = \mathbf{A}^{-1}$, which is the result of the decomposition. For the system in Figure 1, we solve Equation (7) by setting

$$\mathbf{v} = \mathbf{T}^{-1}\mathbf{b} = \begin{pmatrix} 1 & 0 & 0 & & & & \\ \ddots & \ddots & \ddots & & & & \\ & 0 & 1 & 0 & & & \\ c_{m,1} & \cdots & c_{m,m-1} & c_{m,m} & c_{m,m+1} & \cdots & c_{m,2m} \\ & & & 0 & 1 & 0 & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & 0 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_{m-1} \\ b_m \\ \vdots \\ \vdots \\ b_{2m} \end{pmatrix}. \tag{15}$$

This means the vector $\mathbf{v}^k$ can be obtained by manipulating b only at the top and the bottom elements (b_1^k and b_m^k).

If we think of the vector $(\mathbf{z}^k)^T$ as the spikes similar to that in the SPIKE algorithm, our spike can be thought of as two spikes pointing away from (km, km) horizontally, unlike the spikes of the RPDD and SPIKE algorithms, which are column spikes (vertical).

2.3. Cost Savings

The key to the efficiency of the proposed algorithm is based on the decay of the matrix $\mathbf{C}$. By exploiting this decay, we truncate scalar product $\mathbf{z}^k \cdot \mathbf{b}$ to $2J$ terms. This truncation introduces an approximation to $\widetilde{x}_m^k$, which is then transferred to $\mathbf{f}^k$ and propagates to the solutions in the inner domain. Letting $\mathbf{e}^k = \mathbf{f}^k - \widetilde{\mathbf{f}}^k$, the kth subsystem of $\mathbf{A}$ then takes the following form:

$$\mathbf{N}^k\mathbf{x}^k = \mathbf{b}^k - \widetilde{\mathbf{f}}^k - \mathbf{e}^k. \tag{16}$$

The components of the vector $\mathbf{f}^k$ are zero except for the first and the last components, which are given by $f_1^k = l_m^{k-1}\widetilde{x}_m^{k-1}$ and $f_m^k = -\widetilde{x}_m^k + b_m^k$. The vector $\mathbf{e}^k$ represents the errors in the solution due to the approximation at the top. It should be kept in mind that the elements of $\mathbf{e}^k$ are non-zero only in the first and the last components. The interface-splitting algorithm neglects the error term $\mathbf{e}^k$ and solves the following system:

$$\mathbf{N}^k\widetilde{\mathbf{x}}^k = \widetilde{\mathbf{v}}^k = \mathbf{b}^k - \widetilde{\mathbf{f}}^k. \tag{17}$$

As mentioned earlier, it is not necessary to find the actual inverse of the matrix. It is sufficient to compute $\mathbf{z}^k$ from $\mathbf{A}^T\mathbf{z}^k = \widehat{\mathbf{g}}_{km}^k$. This way, the inversion of the matrix $\mathbf{A}$ is avoided. A further savings can be achieved by reducing the matrix $\mathbf{A}$ to a submatrix $\mathcal{A}$, enclosing the interface.

To summarize, the ITS algorithm consists of seven steps described in the following algorithm.

The Algorithm

Letting J and L be some small positive integers, then the interface splitting algorithm is defined as follows (Algorithm 1):

Algorithm 1 The interface splitting algorithm.

1: On p_k $(1 \leq k \leq p)$, allocate $\mathbf{N}^k$, $\mathbf{b}^k$, $\mathbf{x}^k$ and l_1^k.
2: On p_k $(k \leq (p-1))$,
   1. allocate submatrix $\mathcal{A} = a_{ij}$, $(km - J - 2L + 1) \leq i, j \leq (km + J + 2L)$, where $a_{ij} = \mathbf{A}$;
   2. allocate $\widehat{\mathbf{g}'} = \widehat{g}_{km,i}$, $(km - J - 2L + 1) \leq i \leq (km + J + 2L)$, where $\widehat{g}_{km,i}$ is the ith element of $\widehat{\mathbf{g}}_{km}$;
   3. solve $\boldsymbol{\mathcal{A}}\mathbf{z'^k} = \widehat{\mathbf{g}'}$ (Equation (11));
   4. allocate $\mathbf{s}^k = z_j'^k$, $(km - J + 1) \leq j \leq km$;
   5. allocate $\mathbf{u} = z_j'^k$, $(km + 1) \leq j \leq (km + J)$ and send it to p_{k+1}.
3: On p_k $(2 \leq k)$,
   1. receive $\mathbf{u}$ from p_{k-1} and store it in $\mathbf{t}^k$.
4: Compute parts of the solution at the interface.
   1. On p_k $(k \leq (p-1))$, $a_b^k = \mathbf{s}^k \cdot \mathrm{b}_b^k$;
   2. On p_k $(2 \leq k)$, $a_t^k = \mathbf{t}^k \cdot \mathrm{b}_t^k$,

   where b_t^k and b_b^k are the first and the last J elements of $\mathbf{b}^k$, respectively.
5: Communicate the results.
   1. On p_k $(k \leq (p-1))$, send a_b^k to p_{k+1} and receive a_t^{k+1} from it.
   2. On p_k $(2 \leq k)$, send a_t^k to p_{k-1} and receive a_b^{k-1} from it.
6: Modify the rhs.
   1. On p_k $(k \leq (p-1))$, $\widetilde{v}_m^k = a_b^k + a_t^{k+1}$;
   2. On p_k $(2 \leq k)$, $\widetilde{v}_1^k = b_1^k - (a_t^k + a_b^{k-1}) l_1^k$.

   Note: $\widetilde{\mathbf{v}}_m$ is stored in place of $\mathbf{b}^k$.
7: On p_k $1 \leq k \leq p$, solve $\mathbf{N}^k\mathbf{x}^k = \widetilde{\mathbf{v}}^k$ (Equation (14)).

In this algorithm, J is the number of terms that we chose to truncate the scalar product. *The interface-splitting algorithm is equivalent to the direct method up to the machine's accuracy (ϵ) if $\mathbf{z}^k$ decays below ϵ within this truncation length, i.e., $z_i^k < \epsilon$ for $|i - km| > J$.*

Supposing the J was chosen and the omitted terms in the scalar product are not larger than ε_c, then we do not need to compute $\mathbf{z}^k$ to full precision. In order to maintain the accuracy of the smallest terms retained for the scalar product, we need to solve a larger system containing the (km)th row. In the second step, we chose $\mathcal{A}$, whose dimension is $2(J + L)$, with $L < J$. The choice of L and its effects on the accuracy of $\mathbf{z}'^k$ will be explained later.

One of the most important features of the ITS algorithm (Algorithm 1) lies in step 4. Instead of using one-way communication in two sequential steps as in the RPDD algorithm, the ITS algorithm communicates in only one step. The communications with the two nearest neighbors are allowed to overlap. Therefore, the cost of the communication can be reduced by up to 50% with the ITS algorithm, depending on the actual capability of the interconnection link and the network topology.

The error of the interface-splitting algorithm consists of the approximation error of x_m^k and the propagation of this error into the solutions in the inner domain. The factors determining the accuracy of this approximation are: *(i) the row diagonal dominance factor* $\sigma_i = |d_i|/(|l_i| + |r_i|)$ and *(ii) the matrix bandwidth* $2J$ *used to compute* $\widetilde{x}_m^k$. The complexity and the effect of σ on L and J are described in the next section.

2.4. Comparison to Other Approaches

Our approach exploits the decay of the LU decomposition of the system. This property has been exploited many times already in the literature. The low complexity of the RPDD algorithm is achieved by the same means, apart from RPDD [27] and the truncated

SPIKE [32], in which the amount of data transfer is optimal. McNally, Garey, and Shaw [34] developed a communication-less algorithm. In this approach, each processor p^k holding the subsystem $\mathbf{x}^k$ solves a larger system enveloping this subsystem. This new system has to be large enough such that the errors propagated into the desired solution $\mathbf{x}^k$ due to the truncation of the system can be neglected. It can be shown that, for the same level of accuracy, this approach requires J times more data transfer than the proposed algorithm. This high communication overhead and the increase in the subsystem size make this algorithm non-competitive for multiple-rhs problems. A similar concept has been applied to compact finite differences by Sengupta, Dipankar, and Rao in [35]. However, their algorithm requires a symmetrization that doubles the number of floating point operations. It should be noted that the current truncated SPIKE implementation utilizes bidirectional communication as well. However, this improvement is not made public. It will be shown later that such a simple interchange to the communication pattern can greatly improve the performance of parallel algorithms. There are a number of algorithms using biased stencils that mimic the spectral transfer of the inner domain [36–38]. In this approach, the user has no control over the error of the parallel algorithm. Even though the local truncation error of the approximation at the boundary is matched with that of the inner scheme, the discrepancies in spectral transfer can lead to a notable error at the interface in under-resolved simulations.

3. Complexity, Speedup, and Scalability

The complexity of the interface-splitting algorithm depends on two parameters, the one-digit decay length L and the half-matrix bandwidth J. These numbers are determined by the cut-off threshold ε_c and the degree of diagonal dominance of the system ($\sigma = min\{\sigma_i\}$). For Toeplitz systems $\mathbf{T} = [1,\ \lambda,\ 1]$, the entries of the matrices in the LU-decomposition of this system converge to certain values, and the half-matrix bandwidth J can be estimated from them. Bondeli [39] deduces this convergence and estimates J with respect to the cut-off threshold ε_c by

$$J = -\frac{ln\ \varepsilon_c}{ln\left(\frac{1}{2}\left(|\lambda| + \sqrt{(\lambda^2 - 4)}\right)\right)}. \tag{18}$$

For example, J equals 7 and 27 for $\varepsilon_c = 10^{-4}$ and 10^{-15} when $\lambda = 4$ ($\sigma = 2$), which corresponds to the fourth-order compact differentiation [5] and the cubic spline interpolation. These two numbers are much smaller than the usual size of the subsystem used in scientific computing. Effects of diagonal dominance on J are shown in Figure 2. The proposed algorithm is thus not recommended for a very small σ. This includes the inversion of the Laplace operator. For solving such systems, we refer the reader to the work of Bondeli [26,39], in which he adapts his DAC algorithm for the Poisson equation. In the fourth-order compact schemes, the diagonal dominance of the first derivative and the second derivative are 2 and 5, respectively. In sixth-order compact schemes, the values are 1.5 and 2.75, respectively. Therefore, the proposed scheme is well-suited for these approximations. On the contrary, if the system is not diagonally dominant, J will decay linearly to zero at half the size of the global system e.g., $pm/2$. In such a system, the proposed algorithm will perform poorly. It will be beneficial to combine the ITS and RPDD algorithms in this situation. Using ITS as the pre-processor will reduce the amount of back-correction in the RPDD. This approach will have better cache coherence, in contrast to computing the full correction in any single method by going from one end of the system to the other.

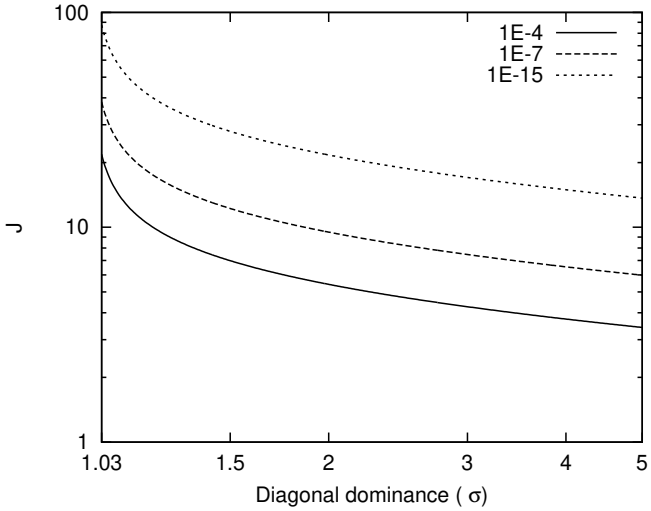


Figure 2. Effects of diagonal dominance on J from Equation (18). The algorithm has a relatively small J for $\sigma > 1.2$ before the effects of the singularity come in.

3.1. Complexity

The interface-splitting algorithm is an approximate method, but it can be made equivalent to other direct methods by setting ε_c to machine accuracy. The minimum decay rate of non-symmetric matrices has been developed in [31,32] and can be used to approximate J. However, the matrix bandwidth obtained in this way is usually too pessimistic for the computation to be efficient because the minimum decay rate depends on σ, which may not be in the vicinity of the interfaces. In multiple-rhs problems, it is worth solving the subdiagonal matrix consisting of the two subsystems enclosing the kth interface and then choosing J according to the desired cut-off threshold. In single-rhs problems, backward elimination of $\mathbf{D}^{k-1}$ and forward elimination of $\mathbf{D}^{k}$ could be an effective way to determine the appropriate J.

In the second step of the algorithm, we have to solve a linear system for the kmth row of the matrix $\mathbf{C}$. This system has to be larger than $2J$ such that the truncated $\mathbf{z}^k$ is sufficiently accurate. In this work, we solve a system of size $2J + 2L$, which ensures that the smallest element of the truncated $\mathbf{z}^k$ is at least correct in the first digit. Since it is unlikely that one would be satisfied with errors larger than 10^{-4}, we assume that $L = J/4$ and use this relation to report the operation count of the ITS algorithm in Table 1. A cut-off threshold lower than this leads to lower complexity. Thus, we assume $L = J/4$ reflects the practical complexity in general applications. In this table, we also list the communication time, which can be expressed by a simple model: $\tau_{com} = \alpha + \beta\Delta$, where α is the fixed latency, β is the transmission time per datum, and Δ is the number of data. Note that, for multiple-rhs problems, we neglect the cost of the first to the third step of the algorithm because the number of rhs is considered to be much larger than m.

In Table 1, the complexity of the proposed algorithm is presented, assuming that each processor only has knowledge of its subsystem and that they do not know the global system. The coefficients thus have to be sent among the neighbors, leading to a higher data transfer in single-rhs problems. Otherwise, the communication is reduced to that of the multiple-rhs problem.

Table 1. Computation and communication costs of the interface-splitting algorithm. The coefficients of the system are assumed to be known only to the owner processor. For multiple-rhs problems, the cost in the first to the third step is neglected. The symbol γ stands for the number of rhs, α is the latency (seconds per communication) of the bandwidth, and β is the data bandwidth (operands per second).

System	Matrix	Sequential	Interface-Splitting Algorithm	
			Computation	Communication
Single-rhs	Non-periodic	$8n-7$	$8\frac{n}{p}+24J-11$	$2\alpha+19\beta J/4$
	Periodic	$14n-16$	$8\frac{n}{p}+24J-11$	$2\alpha+19\beta J/4$
Multiple-rhs	Non-periodic	$(5n-3)\gamma$	$\left(5\frac{n}{p}+4J-4\right)\gamma$	$\alpha+\beta\gamma$
	Periodic	$(7n-1)\gamma$	$\left(5\frac{n}{p}+4J-4\right)\gamma$	$\alpha+\beta\gamma$

3.2. Speedup and Scalability of the Algorithm

Using Table 1, we can estimate the absolute speedup of the ITS algorithm. The absolute speedup here means the solution time on a single processor from the fastest sequential algorithm divided by the time used by the ITS algorithm on p processors. In theoretical complexity analysis, one usually assumes a constant value of communication bandwidth. In practice, the interconnection network is not completely connected, and the computing nodes are usually linked by less expensive topologies such as tree, fat tree, hypercube, array, etc. The data bandwidth is thus a function of the number of processors as well as the amount of the data being transferred. Let the communication cost be a function of p and γ, that is $\beta=\chi(p,\gamma)$, and assume that n and γ are sufficiently large such that the latency and $1/n$ can be neglected. Then, the absolute speedup of the ITS algorithm in multiple-rhs problems is given by Equation (19) below:

$$\mathcal{S}_\gamma(p)=\frac{p}{1+0.8\frac{J}{m}+0.2\frac{\rho\chi(p,\gamma)}{m}}, \tag{19}$$

where ρ is the peak performance of the machine. The key numbers determining the performance of the interface-splitting algorithm are thus the ratios J/m and $\rho\chi(p,\gamma)/m$. The first one is the computation overhead, and the latter is the communication overhead. If the ratio J/m is small and χ is negligible, an excellent speedup can be expected, otherwise the algorithm suffers a penalty. For example, the absolute efficiency will drop from 92% to 56% when J is increased from $0.1m$ to m on multiple right-hand side systems.

In general, one must size the subsystem appropriately such that the speedup from load distribution justifies the increased complexity, overhead, and communication costs. In the previous speedup equation (Equation (19)), the function $\chi(p,\gamma)$ is the inverse of the data transfer rate, and the product $\rho\chi(p,\gamma)$ is simply the ratio of the peak performance (operation/s) to the data transfer rate (operand/s), similar to the one mentioned in the introduction. Therefore, as long as J is smaller than $4\rho\chi(p,\gamma)$, the overhead of the ITS algorithm will be less than the cost of the communication. For example, let us consider a cluster consisting of older-generation hardware such as dual AMD EPYC 7302 nodes connected by a 100 G network. The value of $\rho\chi(p,\gamma)$ is 983. During the latency time alone, which is (1 μs), the processor could have solved a compact differentiation on a 67^3 Cartesian grid. We already discussed this number in modern computers, and it is much higher than this value. Thus, the $4J$ overhead of the ITS algorithm is very small.

According to Equation (19), our algorithm is perfectly scalable because the speedup does not explicitly contain the number of processors, except for the one hidden in χ. The cost of communication per datum $\chi(p,\gamma)$ is determined by the latency, the data bandwidth, and the topology of the interconnection networks among the computing nodes

and the pattern of communication. When p and γ are large, χ can be the major cost of the computation. Even if the exact formula for χ is not known, we can predict when the algorithm will be scalable for the scaled problem size (fixed γ and m). Using the costs from Table 1 and substituting $\chi(p,\gamma)$ for β, it is easy to see that the proposed algorithm is scalable, i.e., $(S_\gamma(p+1) > S_\gamma(p))$ when $\partial\chi/\partial p > 0$. This means that, as long as the interconnection network is scalable, the ITS algorithm is scalable.

Note that the algorithm assumes $J \leq m$; if this is not true, the program adopting this algorithm should issue a warning or an error to the user. The user then can decide whether to adjust the size of the subsystem or extend the scalar product by including more processors.

4. Accuracy Analysis

In the previous section, we discussed the complexity and established the half-matrix bandwidth J for the selected cut-off threshold ε_c. In this section, we analyze how the cut-off threshold propagates into the solution in the inner variables.

Solving Equation (16) for $\mathbf{x}^k$ is equivalent to solving the $(m+1)\times(m+1)$ linear system

$$\mathbf{F}^k\mathbf{u}^k = \boldsymbol{\omega}^k \tag{20}$$

for $\mathbf{u}^k = [x_m^{k-1}\ \ (\mathbf{x}^k)^T]^T$ with

$$\mathbf{F}^k = \left(\begin{array}{c|c} 1 & \mathbf{0}^T \\ ine\mathbf{w}^k & \mathbf{N}^k \end{array}\right),\ \boldsymbol{\omega}^k = [x_m^{k-1}\ b_1^k\ b_2^k\ \cdots\ b_{m-1}^k\ x_m^k]^T, \text{ and } \mathbf{w}^k = [l_1^k\ 0\ \cdots\ 0]^T.$$

Due to the structure of $\mathbf{N}^k$, it follows that

$$\mathbf{F}^k = \left(\begin{array}{c|c|c} 1 & \mathbf{0}^T & 0 \\ ine\boldsymbol{\psi}^k & \boldsymbol{\Pi}^k & \boldsymbol{\xi} \\ ine0 & \mathbf{0}^T & 1 \end{array}\right),$$

where $\boldsymbol{\psi}^k = [w_1^k\ 0\ \cdots\ 0\ ]^T$ and $\boldsymbol{\xi}^k = [0\ \cdots\ 0\ r_{m-1}^k]^T$. The vector $\mathbf{0}$ is a column vector of zero. Its size should be clear from the context. The interface-splitting algorithm injects an approximation in place of x_m^{k-1} and x_m^k, changes the rhs to $\widetilde{\boldsymbol{\omega}}^k = [\widetilde{x}_m^{k-1}\ b_1^k\ \cdots\ b_{m-1}^k\ \widetilde{x}_m^k]^T$, and solves

$$\mathbf{F}^k\widetilde{\mathbf{u}}^k = \widetilde{\boldsymbol{\omega}}^k, \tag{21}$$

instead of the original system. This process introduces the following error into the solution:

$$\mathbf{h}^k = \widetilde{\mathbf{u}}^k - \mathbf{u}^k.$$

It follows that the error in the solution vector satisfies

$$\mathbf{F}^k\mathbf{h}^k = [\widetilde{x}_m^{k-1}\ \ (\widetilde{\mathbf{x}}^k)^T]^T - [x_m^{k-1}\ \ (\mathbf{x}^k)^T]^T, \tag{22}$$

$$\mathbf{F}^k\mathbf{h}^k = \left(\begin{array}{c} e_m^{k-1} \\ \mathbf{0} \end{array}\right) + \left(\begin{array}{c} \mathbf{0} \\ e_m^k \end{array}\right). \tag{23}$$

Equation (23) indicates that the error at the inner indices (h_i^k, $1 < i < m+1$) is a sum of the error propagated from both interfaces. In order to establish the error bound of the ITS algorithm, we first identify the position of the maximum error.

Proposition 1. *Errors of the interface-splitting algorithm for diagonally dominant tridiagonal matrices are maximal at the interfaces.*

Proof. Since the errors of the interface-splitting algorithm h_i^k in the inner domain satisfy Equation (23), that is,

$$d_i h_i^k = -l_i h_{i-1}^k - r_{i+1} h_{i+1}^k,$$

the error of the inner indices, i.e., h_{i-1}^k, $1 < i < m$, is not larger than the maximum error introduced at the interface because

$$\begin{aligned} \left|d_i h_i^k\right| &\leq \left|l_i h_{i-1}^k\right| + \left|r_i h_{i+1}^k\right| \\ \left|h_i^k\right| &\leq \frac{(|l_i| + |r_i|)}{|d_i|} \max(\left|h_{i-1}^k\right|, \left|h_{i+1}^k\right|) \\ \left|h_i^k\right| &\leq |\sigma_i| \max(\left|h_{i-1}^k\right|, \left|h_{i+1}^k\right|) < \max(\left|e_m^{k-1}\right|, \left|e_m^k\right|). \end{aligned}$$

□

In the next step, we assume that a small number ε_c is set as the error threshold and was used to truncate the vector $\mathbf{z}^k$. A small number J is the minimum $j > 0$ satisfying $z_{km\pm j}^k > \varepsilon_c$. The maximum error of the proposed algorithm thus consists of two parts: (i) the truncated terms and (ii) the round-off error in the calculation of the dot product in Step 3. The following theorem states that the sum of these errors is bounded by a small factor of the cut-off threshold.

Theorem 1. *The maximum error of the interface-splitting algorithm $e_{max} = x_k^m - \tilde{x}_m^k$ is bounded by*

$$e_{max} \leq \left[(2 + L)\epsilon + L\varepsilon_c)\right] |\mathbf{b}|_\infty, \tag{24}$$

with the cut-off threshold ε_c of the coefficient vector $\mathbf{z}^k$, the machine accuracy ϵ, and the length L, in which the magnitude of the coefficients is reduced by at least one significant digit.

Proof. First, let us assume that the rhs stored by the machine is exact, and let $\boldsymbol{\mu}^k$ be the error in the coefficient vector $\mathbf{z}^k$ when it is represented by machine numbers $\widehat{\mathbf{z}}^k$; that is, $\boldsymbol{\mu}^k = \widehat{\mathbf{z}}^k - \mathbf{z}^k$. The hat symbol denotes that the number is a machine-accurate numerical value of the respective real number. Let $fl(x)$ be the floating point operation on x. For instance, if ϕ and φ are real numbers, then $fl(\phi) = \widehat{\phi}$. The addition obeys the following inequalities:

$$|fl(\widehat{\phi} + \widehat{\varphi})| \leq |\widehat{\phi} + \widehat{\varphi}| + \epsilon \leq |\widehat{\phi}| + |\widehat{\varphi}| + \epsilon. \tag{25}$$

The numerical value obtained for $\widehat{x}_m^k$ by computing $\left(\widehat{\mathbf{z}}^k\right)^T \mathbf{b}$ is thus

$$\widehat{x}_m^k = fl\left(\sum_{j=1}^{n} \left(z_j^k + \mu_j^k\right) b_j^k\right) \tag{26}$$

$$\widehat{x}_m^k = fl\left(\sum_{j=1}^{km} \left(z_j^k + \mu_j^k\right) b_j^k\right) + fl\left(\sum_{j=km+1}^{n} \left(z_j^k + \mu_j^k\right) b_j^k\right) + \delta_1. \tag{27}$$

For general matrices whose inverse does not decay, the error δ_1 is bounded by $\epsilon\left|\mathbf{b}\right|_\infty$, and the error in Equation (27) is given by $\left|\widehat{x}_m^k - x_m^k\right| \leq n\epsilon\left|\mathbf{b}\right|_\infty$. Thus, the result can be inaccurate for a very large n. However, this is not the case for the matrices considered here.

On computers, a floating point number is stored in a limited mantissa and exponent. Adding a small number b to a very large number a will not change a if their magnitude differs from the range of the mantissa. In other words, $fl(\widehat{a} + \widehat{b}) = fl(\widehat{a})$ if $|\widehat{b}/\widehat{a}| < \epsilon$. Since $\mathbf{z}^k$ decays exponentially, there is a smallest number L such that $\left|z_{j-L}^k\right| < \left|\frac{1}{10} z_j^k\right|$ for $j \leq km$

and $\left|z_{j+L}^k\right| < \left|\frac{1}{10}z_j^k\right|$ for $j \geq km$. Thus, let us assume that the machine accuracy ϵ lies in $(10^{-\eta}, 10^{-\eta+1})$; for some natural number η, we can rewrite (27) as

$$\widehat{x}_m^k = fl\left(\sum_{j=km-\eta L+1}^{km}\left(z_j^k + \mu_j^k\right)b_j^k\right) + fl\left(\sum_{j=km+1}^{km+\eta L}\left(z_j^k + \mu_j^k\right)b_j^k\right) + \delta_2, \tag{28}$$

with $|\delta_2| < \epsilon|\mathbf{b}|_\infty$, provided that $|\mathbf{b}_{(km-L):(km+L+1)}|_\infty = |\mathbf{b}|_\infty$.

Next, we consider the effects of the exponential decay of $\mathbf{z}^k$ on the behavior of the round-off error. Let $\widehat{a}_b^k$ and $\widehat{a}_t^{k+1}$ be the first and the second sum in Equation (28). Due to the decaying of $\mathbf{z}^k$, the round-off errors only affect the result coming from the first L largest terms; thus, the first sum is reduced to:

$$\widehat{a}_t^k = fl\left(\sum_{j=km-\eta L+1}^{km-L} z_j^k b_j^k + \sum_{j=km-L+1}^{km}\mu_j^k b_j^k\right). \tag{29}$$

Since the exponential decay is bounded by a linear decay, we can conservatively approximate the numerical error due to the second sum in Equation (29), bounded by $L\epsilon/2$. Therefore, the error $\widehat{h}_{m+1}^k$ of the solution x_m^k computed by the scalar product with $\varepsilon_c < \epsilon$ is given by

$$|\widehat{h}_m^k|_\infty = |\widehat{x}_m^k - \left(\widehat{a}_t^k + \widehat{a}_b^k\right)|_\infty \leq (L+1)\epsilon|\mathbf{b}|_\infty. \tag{30}$$

We now have the error bound when the full scalar product $\widehat{\mathbf{z}}^k \cdot \mathbf{b}$ was used to approximate x_m^k. It is thus straightforward to show that the error of the interface-splitting algorithm with the cut-off threshold ε_c is

$$|\widehat{\widetilde{h}}_m^k|_\infty = |\widetilde{x}_m^k - x_m^k|_\infty \tag{31}$$

$$\leq \left|\widehat{h}_m^k - fl\left(\sum_{j=1}^{km-J} z_j^k b_j^k\right) - fl\left(\sum_{j=km+J+1}^{n} z_j^k b_j^k\right)\right|_\infty \tag{32}$$

$$\leq \left[(L+2)\epsilon + L\varepsilon_c)\right]|\mathbf{b}|_\infty. \tag{33}$$

□

The application of the ITS algorithm to numerical simulations does not need to be extremely accurate. The cut-off threshold ε_c can be set several digits lower than the approximation errors. The numerical errors of the algorithm would then be dwarfed by the approximation error. Any effort to reduce the numerical error beyond this point would not improve the final solution. In most cases, the approximation errors are much greater than the machine's accuracy. In such situations, the maximum error of the algorithm is bounded by $\varepsilon_c L|\mathbf{b}|_\infty$.

It should be noted that, when one applies the ITS algorithm to approximation problems, the physical requirement of the underlying principle should be respected. For example, if the ITS algorithm is applied to an interpolation problem, the sum of the coefficients should be one to maintain consistency. Likewise, the sum of the differentiation coefficients should be zero. This correction is on the order of ε and can be applied to the two smallest (farthest) coefficients.

5. Results

In this section, we present the results of the proposed algorithm. First, the accuracy of the algorithm is investigated using a single-rhs problem. Then, the performance is evaluated using multiple-rhs problems on four different parallel computers. The scalability

of the interface-splitting algorithm is compared with the ScaLAPACK package and the RPDD algorithm. Finally, the scalabilities of the algorithm are tested on massively parallel machines of up to 2048 processors. The algorithm is implemented in FORTRAN, and the processors communicate via the Message Passing Interface (MPI) library. In all of the tests, we solve the tridiagonal system by using the vectorized kernel modified from LAPACK's DGTSV [40]. The LU-decomposition is computed once. Intel's compiler and the MPI library are used with the -O3-xCORE-AVX512 optimization option.

5.1. Accuracy

5.1.1. Effects of Cut-Off Threshold

Table 2 shows the errors of the interface-splitting algorithm applied to a matrix $[l_i, d_i, r_i] = [1, 4, 1]$ encountered in some approximation problems, such as spline interpolation, compact differentiation [5], and compact deconvolution [41]. In this table, we consider the case of differentiation of $f = \sin(20\pi x)$ on $x = [0, 1]$. The unknowns are placed at $x_i = ih$, $0 \leq i \leq 251$. This problem is solved using three partitions. Table 2 shows that the bound given in Equation (24) is close to the actual error. The smallest J in the table already produces an error smaller than the local truncation error of the differentiation. Note that, in this test, the number of grid points per wave is 12.5, which is a very fine resolution. The smallest J here should be adequate for most simulation-based applications because, in practice, it is difficult to reach this resolution for all relevant scales.

If higher accuracy is needed, the bandwidth J can be extended as required. Figure 3 illustrates that the error is not sensitive to the number of unknowns or to the order of the matrix. The errors of the algorithm are well below the bound given in Equation (24). According to the figure, we can choose J to match the approximation errors in each problem we are solving. It should be noted that, if the J is set too small, the convergence will be saturated at the level close to Equation (24), as shown in Table 2, where the algorithm produces errors about twice that of the machine accuracy, with the two largest values of J.

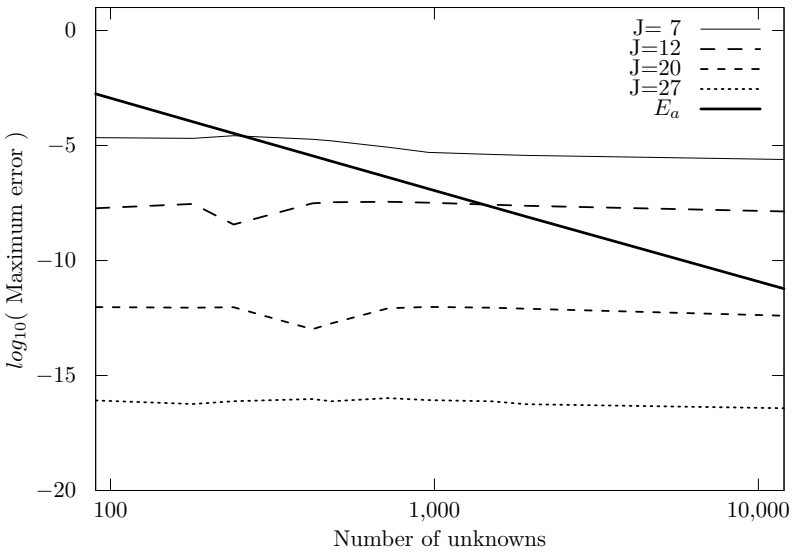


Figure 3. Effect of the order of the system on the accuracy of the interface-splitting algorithm applied to compact differentiation of $\sin(20\pi x)$. The plot shows that the error from the parallelization is approximately constant when n is changed by three orders of magnitude. The approximation error E_a is shown for comparison.

We would like to emphasize that the bandwidth here depends only on the diagonal dominance of the matrix in the neighborhood of the interface and the chosen cut-off threshold ε_c. The number of unknowns does not have a direct influence on this number because the condition number of diagonally dominant matrices converges very fast when the dimension of the matrix is increased.

Table 2. Normalized error of the interface-splitting algorithm applied the fourth-order compact differentiation problem. The error from a different cut-off threshold (ε_c) is compared with the local truncation (E_a) of the differentiation at 12.5 grid points per wave.

J	ε_c	$\frac{1}{\lvert b\rvert_\infty}\lvert\widetilde{f}'_{seq}-\widetilde{f}'_{par}\rvert_\infty$	$E_a=\lvert\frac{f'_{exact}-\widetilde{f}'_{seq}}{f'_{exact}}\rvert_\infty$
7	9.92×10^{-5}	7.13×10^{-6}	
15	2.64×10^{-9}	7.26×10^{-11}	2.77×10^{-5}
27	3.61×10^{-16}	3.85×10^{-17}	

5.1.2. Non-Constant Coefficient Problem

In this section, we investigate the accuracy of the ITS algorithm on a non-constant coefficient problem. In order to create a reproducible test, we set up the matrix using $\mathbf{A} = [\sin(i), 2(|\sin(i)| + |\cos(i)|), \cos(i)]$ and solve a single system of order 1000 with $b_i = 1$ on four processors. The off-diagonal coefficients of this matrix vary relatively fast. The signs change roughly once every three rows. The errors of the ITS algorithm with respect to to the sequential solution are displayed in Table 3, showing that the non-constant coefficients of the matrix do not have negative effects on the accuracy of the algorithm.

Table 3. Normalized error of the interface-splitting algorithm applied to a system with non-constant coefficients with different truncation bandwidth J.

J	7	15	18	20	27
$\frac{1}{\lvert b\rvert_\infty}\lvert x_{seq}-x_{par}\rvert_\infty$	1.4×10^{-5}	2.1×10^{-11}	4.7×10^{-14}	4.4×10^{-16}	4.4×10^{-16}

5.2. Performance

In this subsection, we investigate the performance of the proposed algorithm on three different parallel machines. First, we investigate the scalability of a fixed-size problem by comparing the parallel runtime to the sequential runtime on a single processor. In the second test, we investigate the performance of a scaled problem size. In all tests, a tridiagonal system is solved, with the interface bandwidth J set to 9.

In the first test, we measure a fixed-size speedup on an eight-socket CPU system with Intel Xeon 8168 processors at the Thailand National e-Science Infrastructure Consortium. Each CPU is equipped with three Intel Ultra Path Interconnect (UPI) processors, with a transfer bandwidth of 20.8 GB/s per link per direction. It takes at most two hops for a package to reach the farthest CPU. The problem size in this test is set to 19,200 with 64^3 right-hand sides. This fixed-size speedup measures how much faster a parallel algorithm helps shorten the solution time compared to one CPU. The result is plotted in Figure 4a. Surprisingly, we see a superlinear speedup for $n_p \leq 64$. After $n_p = 64$, the speedup drops below the linear line. This superlinear speedup does not mean the parallel algorithm is better than the sequential one. In a periodic problem, the complexity of our algorithm is indeed lower than the direct Gaussian elimination (see Table 1). However, in this case, our complexity is slightly higher than the sequential code. The superlinear speedup we see here comes from increased memory bandwidth aggregation and better caching. The Xeon processor possesses six memory channels that can transfer the data from the main memory independently in parallel. As the number of processors is increased, the data transfer increases. Our system is an eight-socket with six channels each. Thus, a superlinear speedup up to $n_p = 64$ is reasonable.

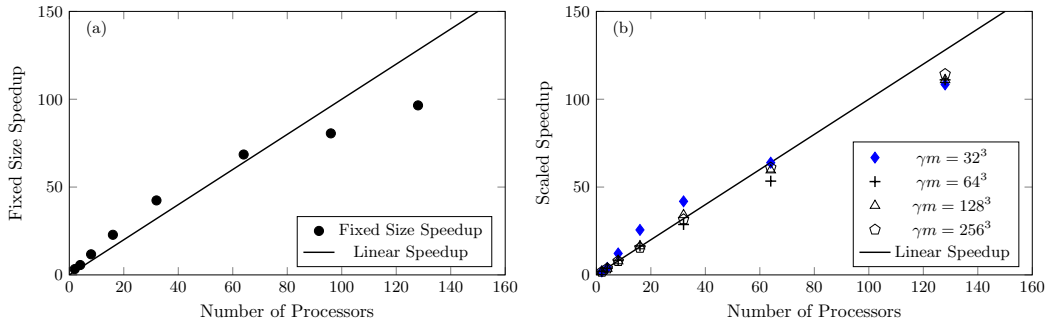


Figure 4. (**a**) Fixed-size speedup (S_γ) of the multiple-rhs problem on an eight-socket Intel Xeon 8168. (**b**) Scaled speedup on the same machine with a number of rhs (γ) equal to the square of the problem size (m).

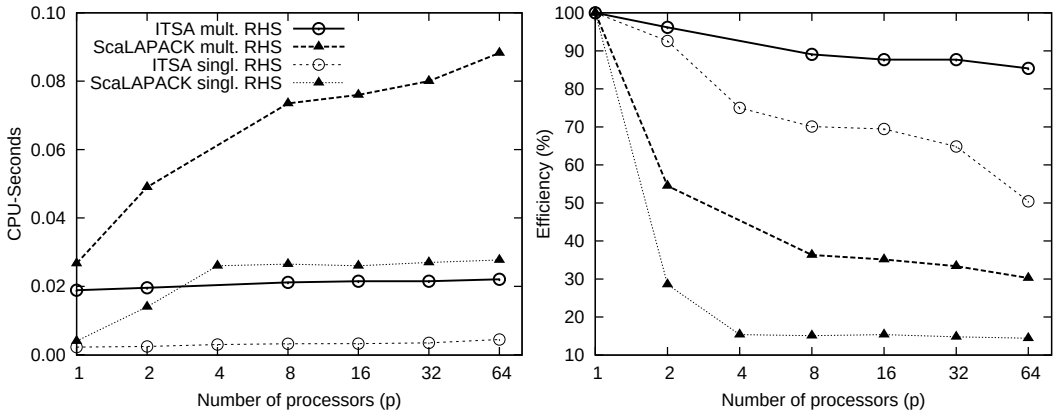
In reality, massively parallel systems are not constructed to solve a problem that can be solved on a single CPU faster. Their actual purpose is to solve a very large system that is impossible to solve on a workstation or on a small cluster. Therefore, the scaled speedup is more relevant to high-performance computing. We conduct a test on the same machine with different problem sizes

We set the number of rhs to $\gamma = m^2$ and measure the scaled speedup, in which each processor solves a system of size m for m^2 right-hand sides. The result is plotted in Figure 4b. We still see a superlinear speedup for the smallest problem size. For other sizes, the scaled speedup is slightly lower than the linear speedup, and the efficiency at the largest number of CPUs is 92% on average.

Comparison with ScaLAPACK

Next, we evaluate the performance of the algorithm on an ALTIX 4700 at the Leibniz-Rechenzentrum LRZ. The scaled speedup of the algorithm is studied for the single- and the multiple-rhs. In this test, the number of unknowns grows linearly with the number of processors, i.e., $n_{total} = pN_{sub}$. The size of the subsystem (N_{sub}) is set to 10^6 in the single-rhs problem. For the multiple-rhs problem, the problem size is 100, and the number of rhs is 10^4. This corresponds to a data block of 100^3 grid points. To highlight the effects of the communications in the solution process, we only report the time used to solve the subsystems (Steps 3 and 4). Likewise, the parallel runtime of ScaLAPACK reported here is the time spent in the PDDTTRS subroutine. The results of the test are shown in Figure 5a. The CPU time of ScaLAPACK is approximately doubled when the number of processors is increased from one to two. This is in accordance with the increase in the complexity of the algorithm [29] used in the ScaLAPACK. Even though the numbers of unknowns in these two problems are equal (10^6), the parallel runtimes of both algorithms are significantly different due to the communications. Interestingly enough, the differences in the CPU time of the multiple rhs problems grow sharply with the number of processors. This behavior indicates that ScaLAPACK is sensitive to the characteristics of the interconnection network and is not as scalable as our algorithm.

The efficiencies of the ITS algorithm and ScaLAPACK are presented in Figure 5b. In the single-rhs problem, the efficiency of the proposed algorithm falls to 50% at $p = 64$ because of the increase in communication overhead, which is half of the parallel runtime there. In the multiple-rhs problem, both algorithms exhibit better performances. At $p = 64$, the ITS algorithm delivers 85% efficiency, compared to 30% of ScaLAPACK. In both problems, the ITS algorithm is at least four times faster than ScaLAPACK.



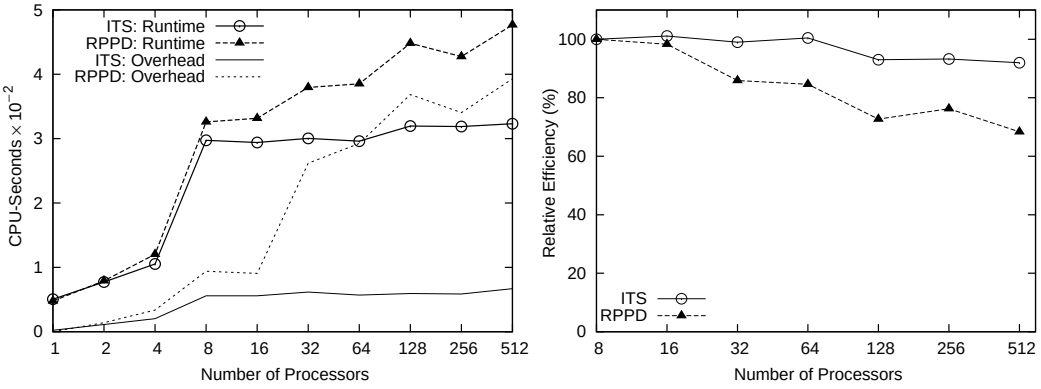
(a) (b)

Figure 5. Parallel runtime (**a**) and the scaled efficiency for (**b**) of the ITS algorithm (solid line) on an ALTIX 4700 compared to ScaLAPACK (dashed line). The problem size for a single-rhs problem (thin line) is 10^6. Subsystem size in multiple-rhs problem is 100, with 10^4 right-hand sides, corresponding to a numerical simulation with 10^3 grid points.

We implemented the RPDD described in [27] and compared the parallel runtime and the scalability in a scaled problem size on the BW-Grid cluster (IBM BladeCenter HS21XM) of the High-Performance Computing Center, Stuttgart (HLRS). In this version, we solve the tridiagonal systems by the same subroutine used in our algorithm. Therefore, the differences between RPDD and our algorithms lie only in the communication pattern. Figure 6a displays the parallel runtimes and the overhead of both algorithms applied to the compact interpolation problem on a 3D Cartesian grid where the size of the subsystem (m) equals 128, with 128^2 right-hand sides. It is not possible to separate the communication time from other overhead in our algorithm because the calculations and the synchronizations are allowed to overlap. Therefore, the overhead of the ITS algorithm here is taken as the total time used in Steps 4–6 of the algorithm described in Section 2. Likewise, the overhead of the RPDD algorithm is the total time used to synchronize and calculate the correction term, including the correction of the computed solution. The parallel runtimes of both algorithms rise sharply from one to eight processors. This increase in the parallel runtime is due to the bandwidth saturation of the bus, which is a typical behavior of shared-memory architecture. The behavior of the parallel runtimes here differs from what is seen in Figure 5 for two reasons: first, because of the density of the node, and second, because of the size of the cache. The computing node on the ALTIX machine used earlier was based on two dual-core processors, but the BW-Grid is composed of two quad-core processors. Both are bus-based shared-memory computers. On the ALTIX 4700, the average cache size per core is 4 MB, which is large enough to accommodate all the right-hand sides. The cache size per core on the IBM machine is only 3 MB, and the right-hand sides do not fit into the cache. The parallel runtimes are thus limited by the data transfer of the memory, which is much slower than the processor speed. After eight processors, the overhead of the ITS algorithms increases slightly, but the overhead of the RPDD algorithm continues to rise very fast. This large overhead is coming from the additional synchronization step. Each processor in the RPDD algorithm must wait for the neighboring processors before the blocking synchronization can be initiated. Therefore, the idle time is potentially doubled, and it eventually leads to a longer overall runtime.

Because the exhaustion of the memory transfer on the computing node can affect the parallel runtime greatly, it is more realistic to measure the scalability relative to the parallel runtime on one computing node rather than the runtime on a single processor. Thus, we plot the relative efficiencies (E_8) in Figure 6b, in which the runtime on the p processor is divided by that of a single node (eight processors). The RPDD algorithm shows good efficiency, but the proposed algorithm is significantly better. At $p = 512$,

the RPDD algorithm delivers a relative efficiency of 63%, while the ITS delivers 92%, which is approximately 50% higher performance.



(a) (b)

Figure 6. Parallel runtime (**a**) and the scaled efficiency relative to eight processors (**b**) of the RPDD and the interface-splitting algorithm on BW-Grid. The sub-problem size is 128 with 128^2 right-hand sides.

5.3. Advantage of Overlapping Bidirectional Communications

The main advantage we expect from our algorithm is a reduction in the communication time during the synchronization phases. The performance is therefore strongly dependent on the implementation of the MPI, which is usually optimized to the size of the data being transferred. In Figure 7a, we present the overall performance of the ITS algorithm on an NEC HPC 144Rb-1 at HLRS with a larger number of processors. The results are gathered from five different runs in which the multiple-rhs system is solved repeatedly 100 times. The same compiler optimization is kept unchanged, and the bandwidth J is set to 10. Each processor holds a set of m^3 data, and for the problem size of m, the number of the right-hand side is m^2.

A sharp rising of the CPU seconds due to memory access is observed in Figure 7a,b for the two largest cases. The parallel runtimes increase by 60% when the number of processors is increased from two to eight processors. Note that this is much better than what was seen earlier in Figure 6a, where the parallel runtime was increased by a factor of six. For $n_p > 8$, the parallel runtime of the ITS algorithm is relatively constant, except for the smallest problem size. The relative efficiencies (E_8) are 50%, 89%, 99%, and 97% from the smallest to the largest problem size, respectively. The speedup in the smallest size suffers from the overhead, which is increased from 40 µs to 400 µs for $p = 8$ to $p = 2048$. When taking into account the minimum time used to solve the system, which is 300 µs, this low efficiency is natural. On the other hand, the parallel runtimes of the RPDD algorithm continue to increase after $p = 8$ as a linear function of p, as shown in Figure 7c,d.

We have examined the ITS algorithm when the non-blocking communications are replaced by the blocking communications (the results are not shown here). This version behaves similarly to the original RPDD algorithm presented above, and the parallel runtime is comparable. Likewise, we have implemented the improved RPDD, which achieves the same performance as our algorithm. In the original algorithm, Sun [27] tried to save computations by having one processor responsible for one interface. This led to the need for two synchronization phases with two blocking communications. See Steps 4 and 5 of the algorithm in Section 2.2 of [27]. However, it is more rewarding to double the computation of that small matrix and have every processor solve both interfaces (top and bottom). There is some redundant work in this approach, but the synchronization in Steps 4 and 5 can be overlapped. This simple change renders the RPDD algorithm as efficient as ours. This gain in the performance comes solely from the reduction in the synchronization step. This role reversal confirms that our version of RPDD is a fair implementation.

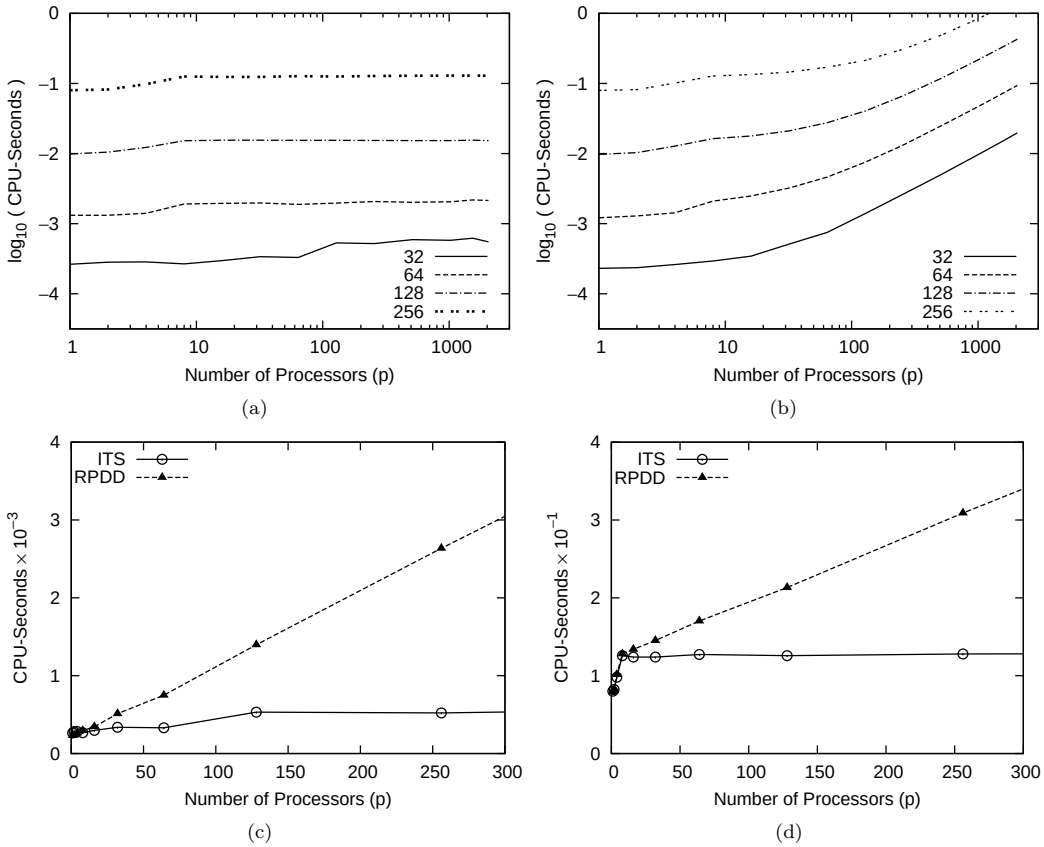


Figure 7. Overview of the parallel runtimes on the Nehalem cluster for different problem sizes of the ITS (**a**) and RPDD (**b**) algorithms. Parallel runtimes on different processors at $m = 32$ (**c**) and $m = 256$ (**d**).

5.4. Application to Navier–Stokes Equations

In this subsection, we present the application of the ITS algorithm to the solution process of the Navier–Stokes equations:

$$\oint_A \mathbf{u} \cdot \mathbf{n} \, dA = 0, \tag{34}$$

$$\frac{\partial}{\partial t} \int_\Omega \mathbf{u} \, d\Omega + \oint_A (\mathbf{u} \cdot \mathbf{n}) \mathbf{u} \, dA = \nu \oint_A \mathbf{T} \, dA - \frac{1}{\rho} \oint_A p\mathbf{I} \cdot \mathbf{n} \, dA. \tag{35}$$

In the following test, the finite volume discretization of the above equations is solved by using compact fourth-order finite volumes described in [7,17,41,42]. First, the two-dimensional inviscid flow is presented, followed a simulation of three-dimensional turbulent channel flow.

5.4.1. Inviscid Doubly-Periodic Shear Layer

This simple 2D flow contains Kelvin–Helmholtz instabilities in which the horizontal shear layer is perturbed by a vertical sinusoidal velocity, leading to a roll-up of the vortex sheet into a cone-like shape. The domain $\Omega = [0, 1]^2$ is taken for this study, and the initial velocities are given by

$$u = \begin{cases} \tanh(\sigma(y - 0.25)) & \text{for y} \leq 0.5, \\ \tanh(\sigma(0.75 - y)) & \text{for y} > 0.5. \end{cases} \tag{36}$$

$$v = \varepsilon \sin(2\pi x). \tag{37}$$

This flow is governed by two parameters: the shear layer width parameter σ and the perturbation magnitude ε. The flow is set to be inviscid ($\nu = 0$) to expose any instabilities the ITS algorithm may cause. In this test, the shear layer parameter and the perturbation magnitude are 30 and 0.05, respectively. To ensure that the approximation error of the ITS algorithm is not affecting the flow, the approximation bandwidth J is set to 12. This will not obtain direct solver-like accuracy, but the parallelization error should be well below the truncation error.

Two grid resolutions are considered for this test case, 512^2 and 1024^2, and the computational domain is equally divided into 4×4 and 8×8, respectively. Figure 8a shows the roll-up of the shear layer, which further reduces the thickness of the layer. Even though the number 512 is illusorily large, it cannot resolve this inviscid flow, and the solution suffers from numerical wiggles. Once the grid resolution is sufficient, the numerical wiggles disappear (Figure 8b). In this fine grid, there are 112 interfaces, and we do not observe any artifacts from the ITS algorithm. Note that the subsystem size of these two simulations is 128^2. At each evaluation of the derivatives in the convection term, the compact scheme costs 6.6×10^5 floating-point operations, while the ITS algorithm costs merely 5.9×10^3—two orders of magnitude smaller. It should be stressed that this case is an inviscid flow, and any numerical artifacts will be amplified. Thus, the approximate nature of the algorithm does not degrade the quality of the simulation nor introduce any numerical artifacts into the solution.

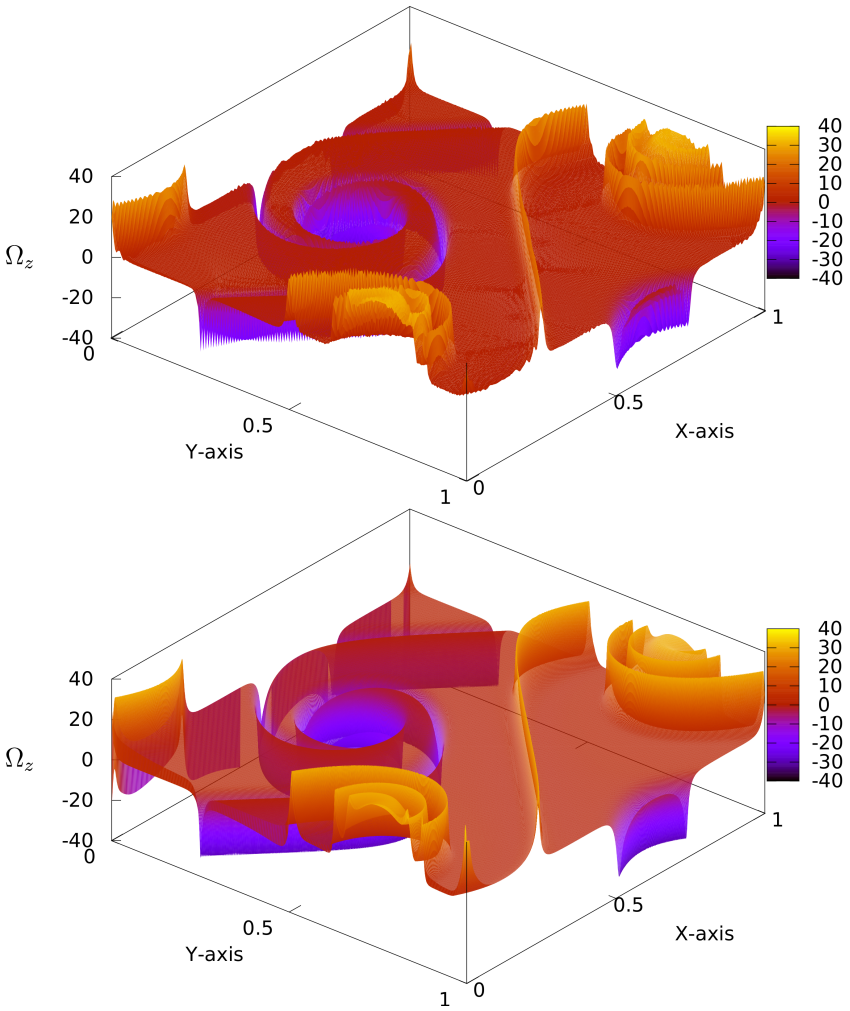


Figure 8. Contoured surfaces of Z-vorticity Ω_z on 512^2 (**top**) and 1024^2 (**bottom**) grids at t = 1.2. The domain is uniformly divided into 4×4 domains on the coarse grid and 8×8 domains on the fine grid.

5.4.2. Turbulent Channel Flow

In this section, we present the result from applying the proposed scheme to a turbulent channel flow at the friction Reynolds number $Re_\tau = 950$. The flow is driven by a constant pressure gradient, and the corresponding bulk flow Reynolds number is 41,430. The computational domain is $[L_x, L_y, L_z] = [2\pi H, \pi H, 2H]$, where the x-axis is the stream-wise direction and z-axis is the wall-normal direction. The numbers of grid points in each direction are $[N_x, N_y, N_z] = [480, 400, 320]$. This problem is solved on $N_{px} \times N_{py} \times N_{pz} \times = 8 \times 8 \times 2$ processors. In the z-direction, the grid is stretched towards the wall.

The mean stream-wise velocity profiles of the fourth-order scheme in Figure 9a collapse on that of the spectral scheme [43]. The r.m.s. of the cell-averaged fluctuations in the span-wise and wall-normal velocities is in excellent agreement (Figure 9b). The position of the peak of u- r.m.s. is correctly predicted. Away from the wall, the prediction of r.m.s. of the stream-wise velocity is slightly lower than what is predicted by the spectral scheme. This difference is due to the averaging procedure, which averages the staggered variables to the location of the pressure cells. This averaging filter removes some of the small scales from the statistics. This shortcoming can be improved by deconvolving the field to the pointwise value.

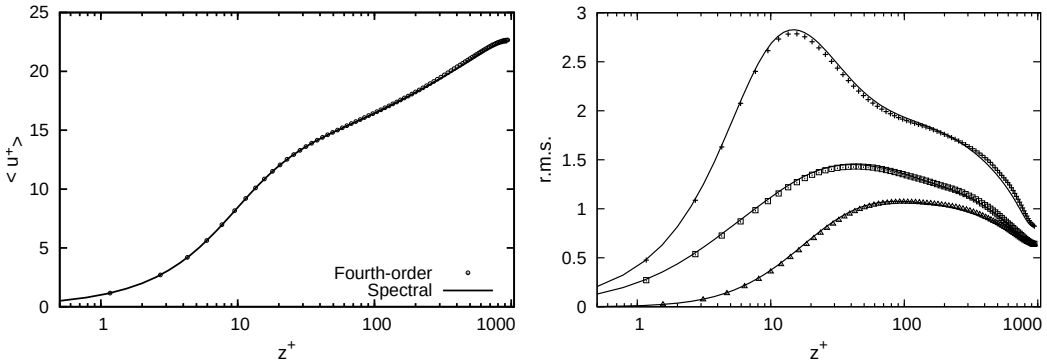


Figure 9. Mean stream-wise velocity (**a**) and r.m.s. of velocity fluctuations (**b**) of the turbulent channel flow ($\mathrm{Re}_\tau = 950$) (every two grid cells are shown).

Finally, a snapshot of the vertical plan $z^+ = 5$ is shown in Figure 10. The plot shows fine detail structures of the wall flow. Small elongated stream-wise streaks are observed. The figure does not show any sign of numerical artifacts. Therefore, the proposed algorithm is accurate and can be used to study real physical phenomena.

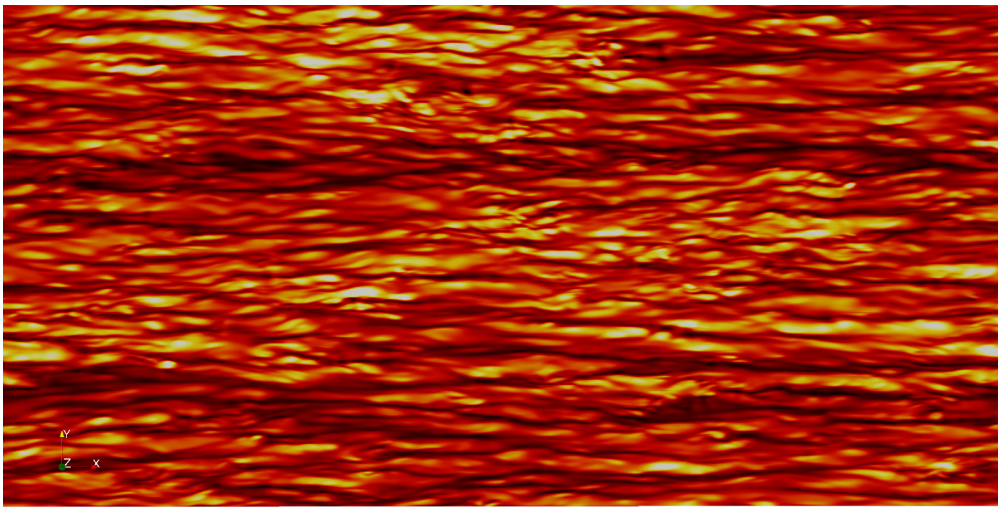


Figure 10. Snapshot of stream-wise velocity at $z^+ = 5$. The contour presents a small streak structure commonly found in wall flow. The black color represents zero velocity, and the white color represents $0.5U_b$. No numerical artifacts are seen in this figure.

6. Conclusions

We have presented the interface-splitting algorithm tailored for diagonal dominant tridiagonal systems. The accuracy and performance of the algorithm rest on the diagonal dominance of the matrix. This algorithm is an approximate method, but it is iteration-less. The user has full control over the accuracy, provided that the subsystem size is sufficiently large. It can be used equivalently to a direct method if desired. Unlike other direct methods, the complexity of this algorithm does not depend on the number of processors, and the leading complexity of the best sequential algorithm is maintained. The algorithm is four times faster than the direct algorithm used in ScaLAPACK. The proposed algorithm is highly efficient and scalable for multiple right-hand side problems.

In massive-scaled computing, the amount of synchronization and overlapping communication is very important. At the same amount of data transfer, the algorithm using overlapping communication can be much faster than the equivalent algorithm using unidirectional communication. The algorithm designer should always check if overlapping communication and a reduction in the synchronization phase are possible. Even though such changes can lead to higher computation costs, the gain from lower communication time can be surprisingly rewarding.

Funding: This work is partially funded by Chiang Mai University.

Data Availability Statement: The data and the source code of the algorithm in this paper available upon request.

Acknowledgments: The author thanks the University of Stuttgart, Leibniz-Rechenzentrum, the Thailand National e-Science Infrastructure Consortium, the Supercomputer Center of Thailand's National Science and Technology Development Agency, and the National Astronomical Research Institute of Thailand for providing computational resources. The author thanks Barbara Wohlmuth and Michael Manhart from Technische Universität München and Carl C. Kjelgaard Mikkelsen from Umea University for fruitful discussions and suggestions during the development of the algorithm and the preparation of this paper.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Pan, G.; Wang, K.; Gilbert, T. Coifman wavelets in 3D scattering from very rough surfaces. In Proceedings of the IEEE Antennas and Propagation Society International Symposium. Digest. Held in Conjunction with: USNC/CNC/URSI North American Radio Sci. Meeting (Cat. No. 03CH37450), Columbus, OH, USA, 22–27 June 2003; Volume 3, pp. 400–403. [CrossRef]
2. Shumilov, B.M. Splitting algorithm for cubic spline-wavelets with two vanishing moments on the interval. *Sib. Èlektronnye Mat. Izv. [Sib. Electron. Math. Rep.]* **2020**, *17*, 2105–2121. [CrossRef]
3. Eren, F.; Gündoğar, Z. Classification of Covid-19 X-ray Images Using Tridiagonal Matrix Enhanced Multivariance Products Representation (TMEMPR). In Proceedings of the 2021 6th International Conference on Computer Science and Engineering (UBMK), Ankara, Turkey, 15–17 September 2021; pp. 221–226. [CrossRef]
4. Pandit, S.K.; Chattopadhyay, A.; Oztop, H.F. A fourth order compact scheme for heat transfer problem in porous media. *Comput. Math. Appl.* **2016**, *71*, 805–832. [CrossRef]
5. Lele, S.K. Compact finite difference schemes with spectral-like resolution. *J. Comput. Phys.* **1992**, *103*, 16–42. [CrossRef]
6. Hokpunna, A.; Rojanaratanangkule, W.; Saedan, M.; Tachajapong, W.; Pherkorn, P.; Manhart, M. Very High-Order Multi-Moment Method for Convection-Diffusion Equation, Part I: Fundamental and Explicit Formulation. *Int. Commun. Heat Mass Transf.* 2023, *submitted* .
7. Hokpunna, A.; Manhart, M. Compact fourth-order finite volume method for numerical solutions of Navier–Stokes equations on staggered grids. *J. Comput. Phys.* **2010**, *229*, 7545–7570. [CrossRef]
8. Hokpunna, A.; Misaka, T.; Obayashi, S.; Wongwises, S.; Manhart, M. Finite surface discretization for incompressible Navier-Stokes equations and coupled conservation laws. *J. Comput. Phys.* **2020**, *423*, 109790. [CrossRef]
9. Pereira, J.M.C.; Kobayashi, M.H.; Pereira, J.C.F. A Fourth-Order-Accurate Finite Volume Compact Method for the Incompressible Navier-Stokes Solutions. *J. Comput. Phys.* **2001**, *167*, 217–243. [CrossRef]
10. Popescu, M.; Shyy, W.; Garbey, M. Finite volume treatment of dispersion-relation-preserving and optimized prefactored compact schemes for wave propagation. *J. Comput. Phys.* **2005**, *210*, 705–729. [CrossRef]
11. Knikker, R. Study of a staggered fourth-order compact scheme for unsteady incompressible viscous flows. *Int. J. Numer. Methods Fluids* **2008**, *59*, 1063–1092. [CrossRef]
12. Yee, H.; Sjögreen, B. Adaptive filtering and limiting in compact high order methods for multiscale gas dynamics and MHD systems. *Comput. Fluids* **2008**, *37*, 593–619. [CrossRef]
13. Ashcroft, G.; Zhang, X. Optimized prefactored compact schemes. *J. Comput. Phys.* **2003**, *190*, 459–477. [CrossRef]
14. Ekaterinaris, J.A. Implicit, High-Resolution, Compact Schemes for Gas Dynamics and Aeroacoustics. *J. Comput. Phys.* **1999**, *156*, 272–299. [CrossRef]
15. Noskov, M.; Smooke, M.D. An implicit compact scheme solver with application to chemically reacting flows. *J. Comput. Phys.* **2005**, *203*, 700–730. [CrossRef]
16. Hokpunna, A. Performance of Sixth-order Finite Surface Method in Turbulent Flow Simulations. *IOP Conf. Ser. Mater. Sci. Eng.* **2019**, *501*, 012044. [CrossRef]
17. Hokpunna, A. Dealiasing strategy for higher-order methods in turbulent flow simulations. *Suranaree J. Sci. Technol.* **2021**, *28*, 1–8.
18. Ishihara, T.; Gotoh, T.; Kaneda, Y. Study of High-Reynolds Number Isotropic Turbulence by Direct Numerical Simulation. *Annu. Rev. Fluid Mech.* **2009**, *41*, 165–180. [CrossRef]
19. Kaneda, Y.; Ishihara, T.; Yokokawa, M.; Itakura, K.I.; Uno, A. Energy dissipation rate and energy spectrum in high resolution direct numerical simulations of turbulence in a periodic box. *Phys. Fluids* **2003**, *15*, L21. [CrossRef]
20. Yokokawa, M.; Itakura, K.; Uno, A.; Ishihara, T.; Kaneda, Y. 16.4-Tflops direct numerical simulation of turbulence by a Fourier spectral method on the Earth Simulator. In Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (Supercomputing '02), Baltimore, MD, USA, 16–22 November 2002; IEEE Computer Society Press: Los Alamitos, CA, USA, 2002; pp. 1–17.
21. Lee, M.; Moser, R.D. Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$. *J. Fluid Mech.* **2015**, *774*, 395–415. [CrossRef]
22. Eidson, T.M.; Erlebacher, G. *Implementation of a Fully-Balanced Periodic Tridiagonal Solver on a Parallel Distributed Memory Architecture*; Technical Report; Institute for Computer Applications in Science and Engineering (ICASE): Hampton, VA, USA, 1994 .
23. Povitsky, A.; Morris, P.J. A higher-order compact method in space and time based on parallel implementation of the Thomas algorithm. *J. Comput. Phys.* **2000**, *161*, 182–203. [CrossRef]
24. Wang, H.H. A Parallel Method for Tridiagonal Equations. *ACM Trans. Math. Softw.* **1981**, *7*, 170–183. [CrossRef]
25. Sameh, A.H.; Kuck, D.J. On Stable Parallel Linear System Solvers. *J. ACM* **1978**, *25*, 81–91. [CrossRef]
26. Bondeli, S. Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations. *Parallel Comput.* **1991**, *17*, 419–434. [CrossRef]
27. Sun, X.H. Application and accuracy of the parallel diagonal dominant algorithm. *Parallel Comput.* **1995**, *21*, 1241–1267. [CrossRef]
28. Arbenz, P.; Gander, W. Direct Parallel Algorithms for Banded Linear Systems. *Z. Angew. Math. Mech.* **1996**, *76*, 119–122.
29. Hegland, M. Divide and conquer for the solution of banded linear systems of equations. In Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, Braga, Portugal, 24–26 January 1996; pp. 394–401.
30. Lawrie, D.H.; Sameh, A.H. The computation and communication complexity of a parallel banded system solver. *ACM Trans. Math. Softw.* **1984**, *10*, 185–195. [CrossRef]

31. Nabben, R. Decay Rates of the Inverse of Nonsymmetric Tridiagonal and Band Matrices. *SIAM J. Matrix Anal. Appl.* **1999**, *20*, 820–837. [CrossRef]
32. Mikkelsen, C.C.K.; Manguoglu, M. Analysis of the Truncated SPIKE Algorithm. *SIAM J. Matrix Anal. Appl.* **2008**, *30*, 1500–1519. [CrossRef]
33. Austin, T.; Berndt, M.; Moulton, D. *A Memory Efficient Parallel Tridiagonal Solver*; Technical Report LA-UR-03-4149; Los Alamos National Laboratory: Walnut Creek, CA, USA, 2004.
34. McNally, J.M.; Garey, L.E.; Shaw, R.E. A communication-less parallel algorithm for tridiagonal Toeplitz systems. *J. Comput. Appl. Math.* **2008**, *212*, 260–271. [CrossRef]
35. Sengupta, T.; Dipankar, A.; Rao, A.K. A new compact scheme for parallel computing using domain decomposition. *J. Comput. Phys.* **2007**, *220*, 654–677. [CrossRef]
36. Kim, J.W.; Sandberg, R.D. Efficient parallel computing with a compact finite difference scheme. *Comput. Fluids* **2012**, *58*, 70–87. [CrossRef]
37. Fang, J.; Gao, F.; Moulinec, C.; Emerson, D. An improved parallel compact scheme for domain-decoupled simulation of turbulence. *Int. J. Numer. Methods Fluids* **2019**, *90*, 479–500. [CrossRef]
38. Chen, J.; Yu, P.; Ouyang, H.; Tian, Z.F. A Novel Parallel Computing Strategy for Compact Difference Schemes with Consistent Accuracy and Dispersion. *J. Sci. Comput.* **2021**, *86*, 5. [CrossRef]
39. Bondeli, S. Divide and Conquer. Parallele Algorithmen zur Lösung Tridiagonaler Gleichungssysteme. Ph.D. Thesis, Eidgenoessische Technische Hochschule Zürich, Zü rich, Switzerland, 1991.
40. Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; et al. *LAPACK Users' Guide*, 3rd ed.; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1999.
41. Kobayashi, M.H. On a Class of Pade Finite Volume Methods. *J. Comput. Phys.* **1999**, *156*, 137–180. [CrossRef]
42. Hokpunna, A. Complexity scaling of finite surface method in high Reynolds number flows. *J. Res. Appl. Mech. Eng.* **2021**, *9*, 1–8.
43. Hoyas, S.; Jimenez, J. Reynolds number effects on the Reynolds-stress budgets in turbulent channels. *Phys. Fluids* **2008**, *20*, 101511. [CrossRef]