Interface Splitting Algorithm for diagonal dominant tridiagonal matrices

Arpiruk Hokpunna, Michael Manhart, Barbara Wohlmuth

Abstract

The interface-splitting algorithm solving diagonal dominant tridiagonal systems in parallel is presented. The algorithm requires only the nearest neighbor communication with minimum number of communications and data transfer. Unlike the existing divide and conquer algorithms solving this problem, the proposed algorithm only requires one phase of synchronisation and bidirectional links can be fully utilised. The operation counts and the accuracy of the algorithm are presented. The performance of the algorithm is evaluated on four different parallel machines. Performance of the ScaLAPACK and the reduced parallel diagonal dominant algorithm of Sun [Parallel Computing, 21:1241-1267, 1995] are presented for comparison. The proposed algorithm is highly scalable and it is at least four-times faster than the ScaLAPACK. At high number of processors, the proposed scheme can be an order of magnitude faster than the reduced parallel diagonal dominant algorithm.

1. Introduction

Solving tridiagonal systems is an important kernel of scientific computing. They appear in many approximation problems such as spline interpolations, wavelets or numerical solutions to differential equations. These systems are usually solved repeatedly for an enormous number of right-hand sides. The best example of such application is the computational fluid dynamics where one solves a three dimensional time-dependent problem. In these problems, the total unknown can easily reach $2^{32}$ which may not fit on a memory of a single processor machine. Efficient parallelisation of these systems are thus critical for scientific computing.

Modern parallel computers are mostly categorised into massive parallel processing (MPP) type which consists of a large number of powerful nodes.
The right-hand-side of the system can be distributed among the processor, and thus a processor holding a certain subsystem may not have explicit access to the subsystems own by other processors. Despite the availability of non-uniform memory access architecture (NUMA) which allows a processor to access the memory of the other processors, this nonlocal access is much more expensive than the local one. This problem is more pronounced when the number of processor does not fit on a single partition in which the computational nodes are connected by special interconnection networks. Efficient and scalable algorithm solving tridiagonal matrices in parallel should minimise the number of communications and synchronisations. Furthermore, a new trend in high-performance computing with Graphical processor units (GPU) requires the parallel algorithm to be even more communication efficient since the data transfer rate per computing power is much more restrictive. In single precision, the data transfer rates from memory to the computing unit are 0.33 data per floating point operation in Itanium2 Montecito and 0.004 on Nvidia GTX285. Coarse grain parallel algorithm such as those in [1, 2, 3, 4, 5, 6] are thus more suitable for the current trend in supercomputer architecture than fine grain parallel algorithm.

Bondeli [3] and Sun [4] independently present the algorithms specialised for diagonal dominant tridiagonal systems. The reduced parallel diagonal dominant algorithm (RPDD) in [4] can solve a tridiagonal system of $n$ equations for $\gamma$ right-hand sides using $(5n/p + 4J + 1)\gamma$ operations on $p$ processors, for some small number $J$ which will be described later. His algorithm is among the most efficient algorithms for this problem. This algorithm requires two times of an unidirectional communication. The bidirectional communication links of modern computers are thus left unused. This shortcoming and problem of load-balancing among the processors can double the cost of the communication.

In the present work, we develop a novel interface-splitting algorithm (ITS) with a complexity of $(5n/p + 4J - 4)\gamma$. This algorithm is designed for diagonal dominant tridiagonal matrices. The idea is to decrease the communication and reduce the data dependency. This is achieved by computing the solution at the interface between two processors before computing the solution on the inner points. This algorithm exploits an exponential decay of the inverse of diagonal dominant matrices demonstrated in [7].

The proposed scheme is competitive. It has less complexity than the algorithm presented in [4] and requires one less synchronisation phase. Therefore the proposed algorithm is less sensitive to load balancing and network conges-
tion problems. This scheme is applicable for non-Toeplitz as well as periodic systems.

This paper is organised as follows. First the interface-splitting algorithm is derived. Then, similarities and differences with existing divide-and-conquer algorithms are discussed and the complexity of the proposed algorithm is presented. Finally, accuracy and performance of the proposed scheme are evaluated on an Ethernet, SGI ALTIX 4700 and NEC HPC 144Rb-1, in comparison to ScaLAPACK and RPDD.

2. Parallel algorithm for solutions of tridiagonal systems

In this section we classify the coarse grained parallel algorithms solving tridiagonal systems in to two types namely, (i) pre-processing and (ii) post-processing algorithms and point out that for an algorithm of each type, it is possible to formulate an equivalent algorithm belong to the other one.

2.1. Pre and post-processing parallel algorithms

We consider tridiagonal systems of size $n$:

$$Ax = b,$$  \hspace{1cm} (1)

where $A$ is a strictly diagonal dominant matrix: $A = [l_i, d_i, r_i]$, $|d_i| > |l_i| + |r_i|$ and $l_1 = r_n = 0$. In order to solve this system in parallel, one can assume that there is a simpler matrix $Q$ and a perturbed right-hand side $v$ with the accompanied transformation matrix $T$ such that

$$Tv = b,$$  \hspace{1cm} (2a)

$$Qx = v.$$  \hspace{1cm} (2b)

This is equivalent to having the tridiagonal matrix decomposed in the form $A = TQ$. The structure of the algorithm depends on the choice of the matrix $Q$. Wang’s partitioning algorithm [1] and parallel line solver [8] belongs to this type of factorisation. This factorisation can be considered as pre-processing schemes where the rhs is perturbed such that the solution of the simpler block matrix gives the desired solution.

On the other hand, one can assume a decomposition in the form $A = Q'S$ and solve

$$Q'w = b,$$  \hspace{1cm} (3a)

$$Sx = w.$$  \hspace{1cm} (3b)
The earliest algorithm of this type applied to tridiagonal matrices is the SPIKE algorithm [2], [9] and [10]. The subsequent algorithms are the parallel diagonal dominant algorithm (PDD) of Sun [11] and the divide and conquer algorithm (DAC) of Bondeli [3]. Even though all of these algorithms are derived differently, their implementations can be identical. In practice, these three algorithms solve Eq.(3b) by two substeps, and the solution $x$ is obtained by

$$x = w - \Delta w,$$  

(4)

using the correction vector $\Delta w$. Sun further exploits the decay of the correction vector and derives the reduced parallel diagonal dominant algorithm (RPDD) in [4]. These algorithms can be considered as post-processing schemes where the first solution $w$ is corrected by solving Eq.(3b). All of these algorithms use the block subdiagonal matrix of $A$ as $Q'$.

A parallel algorithm solving the tridiagonal system will be efficient if the preprocessing step (Eq.(2a)) and the post-processing step (Eq.(3b)) are easy to solve in parallel. The post-processing step of the algorithms in the previous paragraph solves a block tridiagonal system of $p$ subsystems where each block is a $4 \times 4$ matrix. In case of strictly diagonal dominant systems, and if the subsystem size is sufficiently large, the off-diagonal blocks can be smaller than machine accuracy. The matrix $S$ can thus be reduced to a $p$-block diagonal matrix and it can be solved using only the communications between the nearest neighbour which is a highly desirable property.

One can also develop an algorithm of the preprocessing type having the same advantages with RPDD algorithm. This can be easily achieved by solving the preprocessing using $T = S^T$ and use the same block diagonal matrix i.e. $Q' = Q$ in the second step. It is straightforward to show that for any algorithm of the post-processing type, there is an equivalent variant in the pre-processing type using this transformation. There will be small differences in the solution between these two equivalent algorithms due to floating point operations, but it can make significant differences in the programming of the application using these algorithms. For example, if the post-processing algorithm is called from a certain subroutine, the calling routine must have knowledge of the global topology or at least its nearest neighbours. On the contrary when the pre-processing algorithm is used, knowledge of the topology is not necessary and the calling routine can proceed as if it was working alone because solving $Q$ does not require information of the neighbours unlike solving $S$. The pre-processing operations can be done in a higher level
subroutine which has the knowledge of the topology. Therefore the pre-
processing algorithm is more suitable for numerical programs adopting the
domain decomposition concept.

3. The interface splitting algorithm

The interface-splitting algorithm proposed here belongs to the pre-processing
type. For simplicity, we assume that $A$ is a tridiagonal matrix of size $n = pm$
where $p$ is the number of processors and $m$ is some integer which will be
the size of our subsystems. The $k$-th processor is holding the rhs. $b_l$, $(k - 1)m + 1 \leq l \leq km$. Let $D^k$ be the $k$-th block subdiagonal matrix
of $A$, i.e. $D^k = \{ a_{ij} \mid (k - 1)m + 1 \leq i, j \leq km \}$ and $N^k$ be the matrix $D^k$ except the last row being replaced by that of the identity matrix. Instead of
using the block diagonal matrix of $D^k$ as the independent subproblem like
previous algorithms, the interface splitting algorithm uses $N^k$. The matrix
$Q$ and the matrix $N^k$ are illustrated in Fig.1 below.

$$Q = \begin{pmatrix} N^1 & \cdots & N^p \\ \end{pmatrix} \quad \text{and} \quad N^k = \begin{pmatrix} \cdots & \cdots & \cdots \\ l_{c+1} & d_{c+1} & r_{c+1} \\ \cdots & \cdots & \cdots \\ l_{c+m-1} & d_{c+m-1} & r_{c+m-1} \\ \end{pmatrix}$$

Figure 1: Matrix $Q$ of the interface splitting algorithm and its $N^k$ subdiagonal matrix
with $c = (k - 1)m$.

The matrix $Q$ is accompanied by the transformation matrix $T$ which is
just the identity matrix whose $km$-th rows are replaced by a vector $(y^k)^T$. $T$
can be inverted easily thus allowing to solve Eq.(2a) by

$$v = T^{-1}b. \quad (5)$$

It can be shown that the matrix $T^{-1}$ has exactly the same structure as $T$ but
the $km$-th row are changed to $(z^k)^T$ and $z^k_j = c_{km,j}$, $1 \leq j \leq n$ with $C = A^{-1}$. This is equivalent to computing the solution $x_{km}$ explicitly using the $km$-th
row of $A^{-1}$. The solution vector $v$ can be obtained by manipulating $b$ only at the interface i.e. $v = b - f$ using a sparse vector $f$ whose component is non-zero only in the neighbourhood of the interface. The application of this approach for a general matrix is, of course, expensive and prone to numerical instability. However when $A$ is strictly diagonal dominant, the calculations of $z^k$ and $(z^k)^T \cdot b$ are stable and accurate. Nabben has shown in [7] that the components of matrix $C$ decay exponentially away from the diagonal. The key to the efficiency of the proposed algorithm is based on the truncation of the scalar product $(z^k)^T \cdot b$ to a certain bandwidth $2J$ which introduces an approximation error $e^k$ for the rhs. of Eq.(5) (see later).

Thus the interface-splitting algorithm partitions the whole system of Eq.(1) into $p$ smaller independent subsystems. Each subsystem, $x^k$, $x^k_j = x_{(k-1)m+j}$, $1 \leq j \leq m$, is separated from the others by the interface $x^k_m$. This solution at the interface is explicitly computed by a truncated scalar product $\tilde{x}^k_m = (z^k)^T \cdot b$. The dependencies between the subsystems are replaced by these precomputed solutions. The $k$-th subsystem of $A$ then takes the following form

$$N^k x^k = b^k - f^k - e^k. \quad (6)$$

The components of the vector $f^k$ are zero except the first and the last component which are given by $f^k_1 = -l^k_{m-1} \tilde{x}^k_m$ and $f^k_m = -\tilde{x}^k_m - v^k_m$. The vector $e^k$ contains the error of the approximation which is a result of the approximations introduced to the top and the bottom interfaces. Note that $e^k$ are non-zero only in the first and the last components as well. The interface splitting algorithm neglects the error term $e^k$ and solve for the following approximate solution

$$N^k \tilde{\tilde{x}}^k = \tilde{v} = b - f. \quad (7)$$

The interface-splitting algorithm consists of the following four steps:

**step 1. Setting up the subsystems:** The $k$-th processor takes $D^k$ from $A$ and the $x^k$ is distributed to its respective owner. The last row of $D^k$ is replaced by that of the identity matrix to obtain $N^k$.

**step 2. Computation of $z^k$:** Each processor obtains $z^k$ by solving $A^T z^k = r_{km}$ where $r_{km}$ is the vector whose $km$-th component is one and zero otherwise. Assign the coefficient vector $s^k_j = z^k_j$, $(km - J + 1) \leq j \leq km$ and send $z^k_j$, $(km + 1) \leq j \leq (km + J)$ to processor $p_{i+1}$ where it will be stored as $t^k$. These two new vectors will be used to compute the solution at the bottom interfaces.
step 3. Computation of the solution at the interfaces: Compute $a_k^0 = t^T \nu_l$ and $a_1^k = s^T \nu_r$ where $\nu_l$ and $\nu_r$ are the first and the last $J$ components of $\tilde{\nu}^k$, respectively. The impact of $J$ on the accuracy of the algorithm will be explained later. For even processors, send $a_k^0$ to $p_{k-1}$ and receive $a_1^{k-1}$ from $p_{k-1}$. For odd processors, send $a_1^k$ to $p_{k+1}$ and receive $a_0^{k+1}$ from it. The solutions $\tilde{x}_m^k$ at the lower interfaces is given by $\tilde{x}_m^k = a_1^k + a_0^{k+1}$. The right-hand side $\nu_1^k$ of the upper interface is thus $\nu_1^k = b_1^k - (a_0^k - a_1^{k-1}) \nu_l^k$.

step 4. Parallel solution of $x^k$: Each processor can solve its own independent subsystem: $N_k x^k = b^k$. Note that the solution at the last line can be skipped as it is already known.

Step 1 and 2 are only performed once for each matrix $A$, therefore when we are solving multiple right-hand sides the cost in these two step can be neglected. However, computing $z$ this way requires the knowledge of the global topology. This can be problematic for a complex code where many grid blocks of different sizes can be connected together. Because the components of $z^k$ are exponentially decays, it is implied that the components of $A$ far away from the $k$-th interface only have small influences on $z^k$ and thus can be neglected, for example those $a_{ij} \in \{a_{ij} \mid |i - km| > J + \theta, \exists \theta < J\}$. Therefore it is sufficient to solve for a portion of $z^k$ from a smaller matrix $A^k$ which is a diagonal block submatrix enclosing the desired $k$-th interface. This simplification of step 2 significantly reduces the complexity of the coding and makes the interface-splitting algorithm much more efficient in the case of single right-hand side problem. The selection of safety boundary $\theta$ will be described later.

The interface-splitting algorithm will be equivalent to a direct method up to the machine accuracy ($\epsilon$) if $z^k$ decays below $\epsilon$ within the truncation length i.e. $z_i^k < \epsilon$ for $|i - km| > J$. The theory for the exponential decay of the component of $x_k^i$, for non-symmetric banded matrices is well established in [7]. The Accuracy of the interface splitting algorithm depends on the accuracy of the approximate solution $x_m^k$. The factors determining the accuracy of $x_m^k$ are: (i) the row diagonal dominance factor $\sigma = d_i/(|l_i| + |r_i|)$ and (ii) the approximation bandwidth $J$ used for the computation of $x_m^k$. The complexity and the relationship of $\sigma$ and $J$ are describe in the following section.

As mentioned earlier that our algorithm exploits an exponential decay of the inverse of diagonal dominant matrices. This property has been exploited many times already in the literature for tridiagonal matrices. Apart from the SPIKE algorithm and other variants, Macnally, Garey and shaw [12] developed a communication-less algorithm. Each processors $p$ holding the
subsystem $x^p$ in this algorithm solves $M y = u$ where $M$ is the a larger subdiagonal matrix enclosing $D^p$, $y$ and $u$ are the corresponding unknown and the rhs vectors. Similar concept has been applied to compact finite difference in [13]. These approaches require a significant data transfer and the larger subsystem size also increases the cost of the algorithm as well.

4. Complexity and performance

The complexity of interface-splitting algorithm depends on the magnitude of the cut-off threshold which reflects upon two parameters, the 1-digit decays bandwidth $L$ and the necessary bandwidth $J$ achieving the cut-off threshold. The numbers $J$ and $L$ only depend on two factors, the cut-off threshold and the degree of diagonal dominance of the system ($\sigma = \max \{\sigma_i\}$). For Toeplitz system $T = [1, \mu, 1]$, the entry of the matrices of the LU-decomposition of this system converges to certain values and the bandwidth $J$ can be deduced from these values. Bondeli [14] deduce this convergence and gives the necessary bandwidth $J$ achieving the cut-off threshold $\varepsilon_c$:

$$J = -\frac{\ln \varepsilon_c}{\ln \left(\frac{1}{2}(|\mu| + \sqrt{\mu^2 - 4})\right)},$$

which is also applicable to our algorithm. For example, $J$ equal to 7 and 27 for $10^{-4}$ and $10^{-15}$ when $\mu = 4$ ($\sigma = 2$) which is corresponding to fourth-order compact differentiation [15] and the cubic spline interpolation. These two numbers are much smaller than the usual size of the subsystem used in scientific computing. The interface splitting algorithm is an approximate method but it can be made equivalent to other direct methods by setting $\varepsilon_c$ to machine accuracy. The minimum decay rate of $A^{-1}$ which can be used to approximate the the bandwidth $J$ for nonsymmetric matrix has been developed in [16] and [7]. The $J$ obtained this way is usually too pessimistic for an efficient computation because the minimum decay rate depends on $\sigma$ which may not be in the vicinity of the interfaces. In multiple rhs problem, it is worth to solve $A_{2m \times 2m}^k$ and choose $J$ according to the desired error. In single rhs problem, backward elimination of $D^{k-1}$ and forward elimination of $D^k$ could be an efficient ways to determine the appropriate $J$ and $\theta$ for each interface.

In step 2, we have to solve a linear system for the $km$-th row of the $C$. This system has to be larger than $2J$ such that the coefficients of $(z^k)^T$.
are sufficiently accurate. In this work we solve a system of size \(2J + 2L\) which gives a 2-digits accurate representation of the smallest element in the truncated \((Z^k)^T\). In order to make a complexity concise, we assume that \(L = J/4\) which corresponding to \(\varepsilon = 10^{-4}\). It would be rare that one would satisfy with the error larger than \(10^{-4}\). The cut-off threshold smaller than this leads to a smaller complexity, this assumption \((L = J/4)\) is thus reflects a maximum complexity in general applications.

This value of \(L\) is substituted in to the operation counts of the interface-splitting algorithm and the results are shown in Tab.1. In this table we list also the communication time for \(N\) which can be expressed by a simple model as \(\tau_{\text{com}} = \alpha + \beta Q\) where \(\alpha\) is the fixed latency and \(\beta\) is the transmission time per datum. For the single \(rhs\) systems, we assume that the system is so large (otherwise we would not need a parallelisation) such that each processor only knows their own \(rhs\) and the submatrix \(A^k\). This leads to a higher communication compared to [3] and [4] in which the global matrix is known to all processors. If the same assumption is taken here, the communication will be reduced to \(\alpha + 2\beta\). or multiple right-hand side system, we do not count the computation in the first two steps. This leads to the absolute speedups \(S_1\) and \(S_\gamma\) for single \(rhs\) and multiple \(rhs\) shown in Eq.(9) and Eq.(10), when \(n\) is much greater than \(p\) and the communication cost is small.

\[
S_1 = \frac{p}{1 + 3\frac{J}{m}} \quad \quad (9)
\]

\[
S_\gamma = \frac{p}{1 + 0.8\frac{J}{m}} \quad \quad (10)
\]

The key number determining the performance of the interface-splitting algorithm is the ratio \(J/m\). When this ratio is small, 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 0.1\(m\) to \(m\) on multiple right-hand side system.

In Tab.1, the complexity of the interface splitting algorithm is presented assuming that the global system is composed of the subsystem known to its respective processor but the explicit global system is not known. The coefficients are thus sent among the neighbours and this leads to higher data transfer. If the global system is known to all processor, the communication is of course reduced to that of the multiple \(rhs\) problem. According to Tab.1, the interface-splitting algorithm has the complexity slightly lower than the RPDD algorithm [4] for the multiple right-hand side problem. Our algorithm
also has a lesser complexity for a single right-hand side system, when \( J < 0.375m \). Therefore the interface splitting algorithm is competitive.

It is a common practice that the subsystem size should be reasonably large such that the speedup from load distribution justifies the communications and other overheads. This algorithm assumed that \( J \leq m \), if this does not hold the program adopting this algorithm should issue a warning or an error to the user.

<table>
<thead>
<tr>
<th>System</th>
<th>Matrix</th>
<th>Sequential</th>
<th>Interface-Splitting Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Computation</td>
<td>Communication</td>
</tr>
<tr>
<td>Single ( rhs )</td>
<td>Nonperiodic</td>
<td>( 8n - 7 )</td>
<td>( \frac{8n}{p} + 24J - 11 )</td>
</tr>
<tr>
<td></td>
<td>Periodic</td>
<td>14( n - 16 )</td>
<td>( \frac{8n}{p} + 24J - 11 )</td>
</tr>
<tr>
<td>Multiple ( rhs )</td>
<td>Nonperiodic</td>
<td>((5n - 3)\gamma )</td>
<td>((\frac{5n}{p} + 4J - 4)\gamma )</td>
</tr>
<tr>
<td></td>
<td>Periodic</td>
<td>((7n - 1)\gamma )</td>
<td>((\frac{5n}{p} + 4J - 4)\gamma )</td>
</tr>
</tbody>
</table>

Table 1: Computation and communication costs of the interface-splitting algorithm. The system is assumed to be only known to the owner processor.

In this section, we have established the complexity and the bandwidth \( J \) for the cut-off threshold \( \varepsilon_c \). In the next section, we consider how this cut-off threshold affects the accuracy of the solution.

5. Accuracy Analysis

Solving Eq.(6) for \( x^k \) is equivalent to solving the following \( (m+1) \times (m+1) \) system

\[
F^k u^k = z
\]

(11)

for \( u = [x^{k-1}_m \; x^k \; x^k_m]^T \) with

\[
F = \begin{pmatrix}
1 & 0^T \\
\mathbf{w} & \mathbf{N}^k
\end{pmatrix}, \quad z = [x^{k-1}_m \; b_1 \; b_2 \; \cdots \; b_{n-1} \; x^k_m]^T \quad \text{and} \quad \mathbf{w} = [t^k_1 \; 0 \; \cdots \; 0]^T.
\]

(12)
Due to the structure of $\mathbf{N}^k$, it follows that

$$
\mathbf{F}^k = \begin{pmatrix}
1 & 0^T & 0 \\
0 & \mathbf{w} & \mathbf{G}^k \\
0 & 0^T & 1
\end{pmatrix}.
$$

(13)

The interface splitting algorithm introduces an approximation to $x_{m}^{k-1}$ and $x_{m}^k$ and the system $\mathbf{F}\tilde{u}^k = \tilde{z}^k$ is solved instead using $\mathbf{z} = [\tilde{x}_{m}^{k-1} \ b_1 \ b_2 \ \cdots \ b_{n-1} \ \tilde{x}_{m}^k]^T$. These approximations create errors which can be written as follows

$$
\mathbf{h}^k = \tilde{\mathbf{u}}^k - \mathbf{u}^k
$$

(14)

$$
\mathbf{F}\mathbf{h}^k = \tilde{\mathbf{z}}^k - \mathbf{z}^k
$$

(15)

$$
\mathbf{F}\mathbf{h}^k = \begin{pmatrix}
e_{m}^{k-1} \\
0
\end{pmatrix} + \begin{pmatrix} 0 \\
e_{m}^{k}
\end{pmatrix}
$$

(16)

The error at the inner indices ($h_i^k$, 1 < $i$ < $m + 1$) is thus a sum of the error propagated from both interfaces. The position of the maximum error is given by the following proposition.

**Proposition 1.** Errors of the interface-splitting algorithm for diagonal-dominant tridiagonal matrix are maximal at the interface.

**Proof.** The errors of the interface-splitting algorithm $h_i^k$ satisfy

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

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

$$
|h_i^k| \leq |l_i h_{i-1}^k| + |r_{i+1} h_{i+1}^k|
$$

$$
|h_i^k| \leq (|l_i| + |r_{i+1}|) \max(h_{i-1}^k, h_{i+1}^k)
$$

$$
|h_i^k| < \max(h_{i-1}^k, h_{i+1}^k) < \max(e_{m}^{k-1}, e_{m}^{k}).
$$

$\square$

11
In what follows we assume that the threshold $\varepsilon_c$ was used to truncate the vector $z^k$ and $J$ is the minimum $j > 0$ satisfying $z^k_{km+j} > \varepsilon_c$. The maximum error of the interface-splitting algorithm thus consists of the truncated terms and the round-off error in the calculation of the dot product in step 3. These errors are however bounded by a small factor of the cut-off threshold.

**Theorem 1.** Let $A$ be the matrix of size $n = pm$ mentioned in Eq. (1) and $(z^k)^T$ be the km-th row of the inverse of $A$, which can be used to compute the m-th solution of the k-th subsystem. Then the maximum error of the interface-splitting algorithm is bounded by

$$e_{\text{max}} = \left( 2L + \frac{1}{2} \right) \varepsilon + \left( L - \frac{1}{2} \right) \varepsilon_c |b|_\infty,$$

(17)

where $\varepsilon_c$ is the threshold for the cut-off coefficient, $\varepsilon$ is the machine accuracy and $L$ is the bandwidth in which the magnitude of the coefficients is reduced by one significant digit.

**Proof.** Assuming that the right-hand-side vector can be represented exactly by machine number and let $\mu^k$ be the error of representing the exact $z^k$ by the machine number $\hat{z}^k = \hat{z}^k - z^k$. The hat symbol here denotes the machine accurate numerical value of the respective real number. Let $\text{fl}(x)$ be the floating point operation on $x$, for example let $\phi$ and $\varphi$ be a real number $\text{fl}(\phi) = \hat{\phi}$ and $|\text{fl}(\hat{\phi} + \hat{\varphi})| \leq |\hat{\phi}| + |\hat{\varphi}| + \epsilon$. The numerical computation of $\hat{x}^k_m$ by $(\hat{z}^k)^T b$ is thus given by

$$\hat{x}^k_m = \text{fl} \left( \sum_{j=1}^{n} (z^k_j + \mu^k_j) b^k_j \right),$$

(18)

$$\hat{x}^k_m = \text{fl} \left( \sum_{j=1}^{km} (z^k_j + \mu^k_j) b^k_j \right) + \text{fl} \left( \sum_{j=km+1}^{n} (z^k_j + \mu^k_j) b^k_j \right) + \delta,$$

(19)

with $|\delta| < 2\epsilon$. Because $z^k$ decays exponentially, there is a smallest number $L$ such that $z^k_j < \frac{1}{10} z^k_j$ for $j < km$ and $z^k_j < \frac{1}{10} z^k_j$ for $j > km$. This means that if the machine accuracy $\varepsilon$ is in $(10^{-\eta+1}, 10^{-\eta})$ for some natural number $\eta$, Eq.(19) is then equivalent to

$$\hat{x}^k_m = \text{fl} \left( \sum_{j=km-\eta L}^{km} (z^k_j + \mu^k_j) b^k_j \right) + \text{fl} \left( \sum_{j=km+1}^{km+\eta L+1} (z^k_j + \mu^k_j) b^k_j \right) + \delta,$$

(20)
provided that the infimum norm of the truncated $b$ is not smaller than the original one.

Let $\hat{a}^0_k$ and $\hat{a}^1_k$ be the first and the second sum in Eq.(20). Again, due to the decaying nature of $z^k$, the round-off errors only affect the scalar product on the first $L$ largest terms, thus the first sum is reduced to

$$\hat{a}^0_k = fl \left( \sum_{j=km-\eta L}^{km} z^k_j b^k_j \right) + fl \left( \sum_{j=km-L}^{km} h^k_j b^k_j \right)$$  \hspace{1cm} (21)$$
and because the exponential decay is bounded by a linear decay, we arrive at the following error bound

$$\hat{h}^k_m = \hat{x}^k_m - (\hat{a}^0_k + \hat{a}^1_k)$$  \hspace{1cm} (22)$$
< fl \left( \sum_{j=km-\eta L}^{km} z^k_j b^k_j \right) + fl \left( \sum_{j=km+1}^{km+\eta L+1} z^k_j b^k_j \right) + \left( L - \frac{1}{2} \right) \epsilon + \delta$$  \hspace{1cm} (23)$$
< \left( 2L + \frac{1}{2} \right) \epsilon. $$  \hspace{1cm} (24)$$

We have now the error bound in case of using the full scalar product $(\hat{z}^k)^T b$ to approximate $x^k_m$. It is thus straightforward that error of the truncated scalar product is bounded by

$$h^k_m = \hat{x}^k_m - x^k_m$$  \hspace{1cm} (25)$$
= \hat{h}^k_m - fl \left( \sum_{j=1}^{km-J} z^k_j b^k_j \right) - fl \left( \sum_{j=km+J+2}^{n} z^k_j b^k_j \right)$$  \hspace{1cm} (26)$$
< \left( 2L + \frac{1}{2} \right) \epsilon + \left( L - \frac{1}{2} \right) \epsilon_c |b|_\infty. $$  \hspace{1cm} (27)$$

If one aims to apply the interface-splitting algorithm in simulation-based applications, the cut-off threshold $\epsilon_c$ can be kept at certain digits below the errors of the numerical approximations. These errors of numerical approximations are usually far larger than the machine accuracy thus $\epsilon_c$. In such cases, the maximum error of the algorithm is bounded by $(L - 1/2)\epsilon_c |b|_\infty$. 

13
6. Results

In this section, we present results of the interface-splitting algorithm applied to single and multiple rhs. on three different parallel computers. The algorithm is implemented in Fortran and the processors communicates via Message Passing Interface Library (MPI). First the accuracy of the interface-splitting algorithm is studied for a simple matrix in an approximation problem. In the second step, we evaluate its performance on a Linux cluster. Finally, performance and the scalability of the interface-splitting algorithm is presented and compared with ScALAPACK package and RPDD algorithm.

6.1. Accuracy

Table 2 shows the error of the interface splitting algorithm applied to a matrix \([1, 4, 1]\) encountered in approximation problem such as spline interpolation, compact differentiation [15] and compact deconvolution [17]. In this table we consider the case of differentiation of \(f = \sin(20\pi x)\) on \(x = [0, 1]\). The unknown are placed at \(x_i = ih, \ 0 \leq i \leq 251\). This problem is solved using three partitions. The table shows that the bound given in Eq.(17) is not so far from the actual error as seen in the first and the last row. Sometimes the error can be much smaller than the bound due to the cancellation of the neglected terms. The error of the algorithm using the smallest \(J = 7\) is already much smaller than the error of the local truncation error of the differentiation. If one consider the fact that there are 25 grid points per wavelength in this problem which is already very fine, we expect that the smallest \(J\) here should be adequate for most simulation-based applications. If higher accuracy is required, the band can be expanded as necessary. Note that the bandwidth \(J\) here is dependent only on the diagonal-dominance of the matrix surrounding the interface and the choice of cut-off threshold \(\varepsilon_c\).

The number of unknowns has little influence on the bandwidth \(J\) because the condition number of diagonal dominant matrices converges very fast to a certain small number.

In what follows, we demonstrate the accuracy of the proposed algorithm when applied to a matrix with non-constant coefficient. In order to have a reproducible test, we solve the test matrix \(A = [\sin(i), 2(|\sin(i)| + |\cos(i)|), \cos(i)]\) of order 1000 with \(b_i = 1\) on 4 processors. The off diagonal coefficients of this matrix vary relatively fast. Their signs change approximately once every three rows. The numerical errors of the algorithm shown in Tab.3 indicate
the non-constant coefficients of the matrix do not have adverse effects on the accuracy of the algorithm.

\[
\begin{array}{|c|c|c|c|}
\hline
J & \varepsilon_c & \frac{1}{|b|_\infty} |f'_\text{seq} - f'_\text{par}|_\infty & |f'_\text{exact} - f'_\text{seq}|_\infty \\
\hline
7 & 9.167e-05 & 7.1325e-06 & \\
15 & 2.6350e-09 & 7.2559e-11 & 1.7394e-03 \\
27 & 3.6092e-16 & 3.8095e-17 & \\
\hline
\end{array}
\]

Table 2: Normalised error of the interface-splitting algorithm (column 2) subjected to different cut-off threshold ($\varepsilon_c$). The error of the differentiation (column 3) is shown for a comparison.

\[
\begin{array}{|c|c|}
\hline
J & \frac{1}{|b|_\infty} |x_\text{seq} - x_\text{par}|_\infty \\
\hline
7 & 1.41E-005 \\
15 & 2.06E-011 \\
18 & 4.66E-014 \\
20 & 4.44E-016 \\
27 & 4.44E-016 \\
\hline
\end{array}
\]

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

6.2. Performance

Fixed size speedup of the algorithm is studied by solving a multiple rhs. system of order 25600 and $10^4$ right-hand sides. In all following tests, the interface bandwidth $J$ is set to 9. The absolute fixed size speedup in single precision of the interface-splitting algorithm is performed on a Linux cluster of the Leibniz Computing Center. The Intel’s compiler and MPI library are used with -O3 optimization option. Figure 2 shows an absolute speedup which is defined by the solution time on a single processor using the fastest sequential algorithm (Gaussian elimination) divided by the time used by the interface splitting algorithm on $p$ processors. Surprisingly, we obtain almost an ideal speedup here. At $p = 64$, the speedup is even slightly better than the ideal one. This can be attributed to a better data locality since each processor are now dealing with a smaller data size and hence higher cache...
hit can be expected. Since the proposed algorithm only communicates to the nearest neighbour, the communication cost is fixed and we do not observe any network congestion here.

![Graph showing speedup vs. number of processors](image)

**Figure 2:** Absolute speedup \((S_\gamma)\) of the multiple \(rhs\) problem on Linux cluster.

Next we consider the scalability on Altix4700. The scalability of the algorithm is studied for two types of problem, a single \(rhs\) and a multiple \(rhs\). with the same compiler options used earlier. Here we consider a scaled problem where the total system size grows linearly with the number of processors i.e. \(n_{\text{total}} = pm_{\text{sub}}\). The subsystem size is set to \(10^6\) in a single \(rhs\) problem. For multiple right-hand side problems, it is set to 100 with \(10^4\) right-hand-sides.

The results of the test are shown in Fig.3(a). Even though hardly observable, the runtime of the interface-splitting algorithm grows approximately linear with the logarithm of the number of processors in both types of problem. This can be attributed to the fat-tree topology of the interconnection of the machine. Note that the number of unknowns in both problems are equal i.e. \(10^6\). The difference in runtime seen here came only from the communication time. Here we can not achieved an ideal speedup unlike on the cluster. This is of course a limitation imposed by the hardware. It is obvious that, the ScaLAPACK is not as scalable as our algorithm. The CPU-time is approximately doubled when the number of processor is increased from one to two. This is in accordance to the increase in complexity of the parallel algorithm used by ScaLAPACK. Interestingly the differences in the CPU-
time of the multiple right-hand side problems are increasing sharply when
the number of processors is increased. This indicates that the ScaLAPACK
is more sensitive to the characteristics of the interconnection network than
the proposed algorithm. In Fig.3(b) efficiencies of the interface splitting algo-
rithm and the ScaLAPACK are presented. On a single rhs, the efficiency of
the proposed algorithm falls to 50% at \( p = 64 \) because the increase in com-
munication time that occupies 50% of the CPU-time there. Interestingly,
the increased communication time does not affect the solving time of the
ScaLAPACK (thin dash line) much, especially in the single right-hand side
problem. This is because the increase in the computing time is dominated.
In the multiple rhs problem, both algorithms enjoy better efficiencies. At
\( p = 64 \), the interface-splitting algorithm deliver an 85% efficiency compared
to 30% of the ScaLAPACK. On both problems, the interface-splitting is at
least four-times faster than the ScaLAPACK.

We implement the RPDD described in [4] and compare the runtime and
the scalability on scaled-problem size. The BW-Grid cluster (IBM BladeCen-
ter HS21XM) of the High Performance Computing Center Stuttgart is used
in this evaluation. Figure 4(a) shows the runtime and the overhead of both al-
gorithms applied to the compact interpolation problem on 3D Cartesian grid
with \( k = 128, \gamma = 128^2 \). It is not possible to separate the communication
time from other overhead in our algorithm because the calculation and the
synchronisations are allowed to overlap. Thus the overhead of ITS algorithm

Figure 3: Runtime (a) and the scaled-efficiency for (b) of the interface-splitting algorithm
(solid line) on Altix4700 compared to ScaLAPACK (dash 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.
Figure 4: Runtime (a) and the scaled-efficiency (b) relative to 8 processors of the RPDD and the interface splitting algorithm. The sub-problem size is 128 with $128^2$ right-hand sides.

Here are taken as the total time used in step 3 of section 3. Likewise, the overhead of the RPDD algorithm is the total time used to synchronise and calculate the correction term, including the correction of the computed solution. The runtime of both algorithm on a single node are increased sharply from 1 to 8 processors. This increasing in the runtimes is due to the bandwidth saturation of the bus which is a typical behaviour of the bus-based shared memory architecture. The behavior of the runtimes here differ from what seen in Fig. 3(a) for two reasons. The reason is the density of the node. The ALTIX machine we used earlier was based on four dual-cores processors but the BW-Grid is composed of two quad-cores processors. The second reason is the cache size. On ALTIX 4700, the averaged cache size per core is 4 MB while it is 3 MB per core on the IBM machine. The memory bus and the cache become exhausted and the runtime is limited to the memory transfer rate. After 8 processors, the overhead of the ITS and RPDD algorithms are increased linearly with $\log(p)$ as observed earlier in ALTIX 4700. However the growth rate of the overhead is much higher with the RPDD algorithm.

The higher rate of increasing in the overhead of the RPDD algorithm is attributed to the additional synchronisation phase. Each processor must wait for the neighboring processors before the blocking send-receive can be initiated. In this algorithm, two synchronisations are required at each interface and therefore the idle time is potentially doubled. This eventually leads to the longer overall runtime.

Due to the fact that the exhaustion of the memory bus of the computing
node highly affects the runtime. It is more realistic to measure the scalability relative to the runtime on one computing node rather than the runtime on a single processor. To this end, we plot the relative efficiencies ($E_8$) in Fig. 4(b) which is the absolute efficiency on $p$ processor normalised by that of the single node (8 processors). The RPDD algorithm shows a 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.

6.3. Advantage of overlapping bidirectional communications

Since the main advantage we expect from our algorithm is the reduction of the synchronization phases, the performance is thus strongly dependent on the implementation of the MPI which is usually optimized to the size of the data being transferred. In Fig. 5(a) we present the overall performance in double precision of the ITS algorithm on NEC HPC 144Rb-1. The result are gathered from five different runs in which the multiple rhs system is solved repeatedly for 100 times. The same compiler optimization is kept unchanged and the bandwidth $J$ is set to 10. Each processor holds a set of $k^3$ data and for the problem size of $k$, the number of rhs is $k^2$. We have experimented with different eager limit and the eager limit = 589824 was used to create the graphs in this subsection.

The sharp risings of the CPU-Seconds due to memory access are observed in Fig. 5(a) and Fig. 5(b) for the two largest cases. The runtimes increase by 60% when the number of processors is increased from two to eighth processors. Note that this is much better than what seen earlier in Fig. 4(a) where the runtime was increased by a factor of 590% thanks to the integrated memory controller which scales much better than the bus-based shared memory.

After $p = 8$, the runtime of the ITS algorithm is relatively constant up to 2048 processors except for the smallest case where the communication time becomes dominant at $p = 256$. The runtimes of the RPDD algorithm, on the other hand, continue to increase after $p = 8$ as a linear function of $p$ as shown in Fig. 5(c) and Fig. 5(d). It should be noted that, the eager limit of the MPI variable is very important. On this computer, the default limit is too small for $k = 256$ and the MPI decide to use more expensive algorithm which cause the runtime to reaches the saturation at 0.25s as soon as the task falls out of the node ($p > 8$). On this setting, ITS algorithm only become better than RPDD at $p > 256$. By setting the eager limit larger than the message size (eager limit = 589824), the performance of boths algorithm are
improved. However, ITS algorithm gains more improvement and it is faster than RPDD at every number of processors.

We then examine the ITS algorithm together with blocking communications using two-phase synchronisation (the results are not shown here). This version gives a similar behaviour to RPDD algorithm and the runtime is comparable. We also implement a modified version of RPDD such that only one synchronisation phase is used together with overlapping communications. The performance of this implementation is comparable to that of the ITS algorithm shown earlier. Therefore, the main difference in the runtime between these two algorithms are contributed only to the overlapping communications and the number of synchronisation phases.

Figure 5: Overview of the runtimes on different problem sizes of the ITS (a) and RPDD (b) algorithms. Runtime on different processors of at $k = 32$ (c) and $k = 256$ (d).
7. Conclusion

We have presented the interface splitting algorithm solving diagonal dominant tridiagonal systems. The accuracy of the algorithm depends on the diagonal dominance of the matrix. This algorithm is an approximate method but iteration-less and the user has full control over the accuracy, provided that the subsystem size is sufficiently large. It can be used equivalently to the direct method if desired. Unlike a direct method like ScaLAPACK, the complexity of this algorithm does not depend on the number of processors and the leading complexity of the Gaussian elimination is maintained. The proposed algorithm is evaluated on four different parallel computers. Even though the complexity of the algorithm is independent of the number of the processor, the runtimes show a slight dependency (logarithmic) on ALTIX 4700 due to the network architecture. This algorithm has been shown to be highly efficient and scalable for multiple right-hand side problems. The algorithm is at least four-times faster than the ScaLAPACK which employs a direct algorithm.

It must be strongly emphasise that in a massive-scaled computation, the number of synchronisation and overlapping communication is very important. At the same amount of data transfer, the algorithm using and overlapping communication can be an order of magnitude faster than the equivalent algorithm using blocking communications.

The striking evidence of the advantage of overlapping bidirectional communication is demonstrated on NEC HPC 144Rb-1, where the runtime of ITS algorithm is approximately constant. The runtime of the RPDD, as originally proposed, is however increases linearly with $p$. This eventually leads to an order of magnitude difference in the runtime at large number of processors. Nevertheless, we must stress that, with minor modifications, the RPDD algorithm can deliver a comparable the same performance to our algorithm.


