# A time-optimal algorithm for solving (block-)tridiagonal linear systems of dimension N on a distributed computer of N nodes

#### Martin Neuenhofen

February 2, 2018

#### Abstract

We are concerned with the fastest possible direct numerical solution algorithm for a thin-banded or tridiagonal linear system of dimension N on a distributed computing network of N nodes that is connected in a binary communication tree. Our research is driven by the need for faster ways of numerically solving discretized systems of coupled one-dimensional blackbox boundary-value problems.

Our paper presents two major results: First, we provide an algorithm that achieves the optimal parallel time complexity for solving a tridiagonal linear system and thin-banded linear systems. Second, we prove that it is impossible to improve the time complexity of this method by any polynomial degree.

To solve a system of dimension  $m \cdot N$  and bandwidth  $m \in \Omega(N^{1/6})$  on  $2 \cdot N - 1$  computing nodes, our method needs time complexity  $\mathcal{O}(\log(N)^2 \cdot m^3)$ .

## 1 Introduction

Motivation of the problem Many computational engineering tasks deal with the solution of (systems) of one-dimensional differential-algebraic boundary-value problems [9]. Examples are numerical simulations of the following physical phenomena: the deformation of a clamped beam, the dynamic pressure in a gas-pipe, the trajectory of a missile, and constrained optimal control problems.

Using a numerical discretization method, a large-dimensional equation systems results that is typically solved via Newton's method. The arising linear systems are thin-banded and of very large dimension.

Often, the Newton system results from a minimization principle, either of an objective or by a natural model that aims for minimization of potential energy. In these cases the arising linear systems are not only thin-banded and of very large dimension, but they are also symmetric positive definite, which is clearly desirable for reasons of numerical stability.

Since thin-banded, the system matrix can be interpreted as block-tridiagonal, where the block-size is identical to the band-width. Thus, for ease of presentation in the following we present an algorithm for (block-) tridiagonal systems where the block-size is  $m \ll N$ .

Motivation of the problem statement and computing model In consequence of the emergence of massively parallel computing systems, nowadays the problem in numerical computing is typically not to solve a given mathematical problem, but rather to solve it on a given computing system while exploiting its resources in an optimal way. Especially for parallel computing systems it is difficult to spread the computational task in a way that enables the full use of the computing system's capacity.

As for our case of solving thin-banded linear systems, it is well-known that for a bandwidth bounded by  $m \in \mathcal{O}(1)$  the optimal time complexity for solving a banded linear system on a serial computer is

 $\mathcal{O}(N)$ ,

as can be achieved by use of Gaussian elimination. This is equivalent in meaning to that solving the problem with a serial machine is literally as expensive as the following two: *reading* the problem, or *writing* the problem's solution into the memory.

So at first glance it seems like nothing can be done to improve: The time to communicate the problem to a solver would already predominate the time that the solver actually needed. So where is the point in trying to make the solver faster?

The point or answer is that problems do not need to be communicated. One could assemble in parallel the rows of a linear equation system in a distributed way on the memory of independent computing nodes that are connected via a network. Using the algorithm that we propose, all the computing nodes can solve the one big linear system but each of them does only *read a tiny part* of the problem and only *writes a tiny part* of the solution vector.

Literature review We consider the problem of solving a thin-banded linear system as a generalization of solving tridiagonal linear systems in parallel, which is why our literature review refers to parallel tridiagonal solvers. Such solvers can be applied by interpreting the thin-banded linear system as block-tridiagonal system of dense blocks.

There are several popular methods for the solution of tridiagonal linear systems. These have in common that they use a concept described as *parallel factorizations* [1, 7]: The matrix is multiplied from the left with a block-diagonal matrix that decouples a portion of the unknowns through a n interface system. The reduced system is solved. The reduced solution is distributed to all processors so that they can compute in parallel the formerly removed unknowns.

According to [2], the first parallel tridiagonal solver is called *cyclic reduction* and was presented 1965 in [5]. The *recursive doubling algorithm* was introduced

1973 in [11]. In both of these algorithms each processor holds one row of the system. Cyclic reduction works by successively expressing the odd variables of the solution vector in terms of the even. This can be done until finally there is a system of one variable that is solved. An implementation is provided in [4].

Wang's method [12], introduced 1981, is a parallel algorithm where the order of the number of processors is smaller than the order of the number of rows of the system to be solved. This algorithm has been proven to be numerically stable [13]. The idea of this method is the assembly of an interface problem whose solution can be used directly to solve the remaining variables in a backward-substitution step.

In 1991 Bondeli introduced a divide and conquer algorithm for tridiagonal system [3]. The idea of this algorithm is to solve a block-diagonal approximation of the system in parallel, where each processor holds a diagonal-block. It results a reduced interface system that is solved by cyclic reduction, cf. [2].

Organization of the paper In the remainder of this section we describe the parallel computing system that we use for our algorithm. We then describe the mathematical problem that the algorithm solves. It is important that the problem is given to the algorithm in a special way. In particular, the problem data must be provided in distributed memory before the algorithm is called. This is important because it would take to much time to move the data from a central storage to the distributed memory.

In Section 2 we present the algorithm. We start with an implementation and afterwards show how this algorithm can be derived from familiar matrix algorithms. At the end of the section we analyse the time complexity of the algorithm and remark on optimality of this complexity result.

Section 3 we give lower bounds on the time complexity for solving tridiagonal linear systems on a parallel distributed memory machine. We show that our algorithm is able to yield optimal time complexity.

Eventually we draw conclusions in Section 4.

Computing system We need to describe the computing system before we describe the problem statement because otherwise we cannot describe where we presume the problem data to be placed. Above we described why this is crucial: We have to make sure that the problem is provided in the right way because moving the problem data around would cost too much time.

For a system of dimension  $N \cdot m$  with a bandwidth m we consider a computing system of  $2 \cdot N - 1$  computing nodes that have each their own memory and that each run the solution algorithm in parallel. The nodes are connected via a network of cables. Nodes can send data to other nodes and nodes can wait to receive data from others. Figure 1 illustrates the computing system. The black circles symbolize the computing nodes and the black lines illustrate their connections via a cable network.

For the network we require a special structure: For our algorithm we need a two-tree network. In a two-tree, also called dual tree, each node is connected to three other nodes: a parent, an up-child and a down-child, cf. in the figure. Exceptions are: There exists one node called root. This node does not have a parent. Further, there exist N nodes that are called leaves. A leaf does only have a parent, but it does neither have an up-child nor a down-child.

The right part of the figure assigns the nodes with numbers. Each node has a *processor number* and a *level*. The levels are defined recursively: Each leaf has a level of zero, and the parent of each node has a level that is by one larger than the level of the node itself. The processor number is a value that is given by counting from 1 from the uppermost node to the lowermost node of each level.

Each node holds identifying variables like a passport: The variable my\_level gives the level of this node. The variable my\_proc\_num is a list. The following values are well-defined

$$my\_proc\_num(\ell)$$
 for  $\ell \in \{my\_level, ...d\}$ ,

where d is the level of the root. The value my\_proc\_num( $\ell$ ) gives the processor number of the node on level  $\ell$  through which a signal from the root would have to travel in order to reach this node. Figure 1 gives an example for this: To reach the node on level 0 with processor number 5, a signal from the root would have to traverse through node 2 of level 2 and node 3 of level 1.

We presume the nodes as identical serial computing units. As is common, we presume that basic operations plus, minus, times, divide, and copy of scalar values require each a fixed amount of time on a respective node.

For communications over the network, we use the following message passing interface of six commands:

- send\_to\_up\_child(M). If this node has an up-child then it sends a matrix by copy M to up-child and waits until up-child received it.
- receive\_from\_up\_child(M). If this node has a down-child then it waits until it receives by copy a matrix from up-child. This node stores the matrix in its variable M and the copy for the transmission is destroyed.
- $send\_to\_down\_child(M)$ ; analogous to above, but the data is sent to the down-child of this node.
- $\bullet$  receive\_from\_down\_child(M); analogous to above, but the data is received from the down-child of this node.
- send\_to\_parent(M); analogous to above, but the data is sent to the parent of this node.
- receive\_from\_parent(M); analogous to above, but the data is received from the parent of this node.

For each communication we assume a time complexity of the number of elements of  $\mathbf{M}$  plus a constant amount of time  $c_N^{\mathrm{Lat}}$  that is due to latency. The latency accounts for the phenomenon that information travels through the cable at speed of light, so it takes a while until the beginning of a message has moved through the cable.

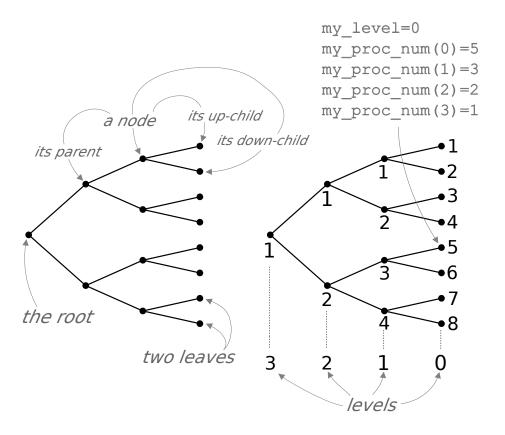


Figure 1: Structure of the computing system:  $2 \cdot N - 1$  nodes are connected in a two-tree communication network. Each node can send messages to its parent and children. The nodes are classified in levels. Further, each node per level is given a processor number. Each node knows its level and the processor number of itself and its parents.

**Problem statement** We consider the numerical solution of a banded linear system

$$\underline{\mathbf{A}} \cdot \underline{\mathbf{X}} = \underline{\mathbf{Y}} \tag{1}$$

where  $\underline{\mathbf{A}} \in \mathbb{R}^{(N \cdot m) \times (N \cdot m)}$  has bandwidth  $b \leq m$ , and  $\underline{\mathbf{Y}} \in \mathbb{R}^{(N \times m) \times k}$  is a dense matrix of k right-hand sides. The task is to find numerical values for  $\underline{\mathbf{X}} \in \mathbb{R}^{(N \times m) \times k}$ . We assume  $N \in 2^{\mathbb{N}}$ .

As formerly discussed, the time for writing the solution into memory in a sequential way would already exceed the time that is actually needed to solve the system. This is why in the following we describe very precisely in which form the data **A**, **Y** must be provided to our computing system.

The system matrix, the right-hand sides, and the solution vectors are stored in a separated way in the leaves of our two-tree. Figure 2 illustrates the situation for N=8. Each leaf holds five matrices in its private storage:  $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{m \times m}$  and  $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m \times k}$ . Matrices of two distinct leaves can have totally different values. Comparing the upper and lower part of the Figure 2, we find that the original matrices  $\underline{\mathbf{A}}, \underline{\mathbf{X}}, \underline{\mathbf{Y}}$  can be composed of the matrices  $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{X}, \mathbf{Y}$  of all leaves. There are two matrices that fall out of the pattern:  $\mathbf{C}$  in the uppermost leaf and  $\mathbf{B}$  in the lowermost leaf. We require that these matrices are zero-matrices.

## 2 The algorithm

This section is organized as follows. We first present the algorithm as a code that could be directly used for implementation in a programming language such as MPI with Cpp or Fortran. Then we sketch our derivation of the algorithm, that arised from applying the devide-and-conquer paradigm on the SPIKE algorithm due to Sameh [8]. We explain how our algorithm operates and give an example to illustrate the algorithmic steps. Finally, we analyse the parallel time complexity.

**The algorithm** The following algorithm is launched on all nodes of the computing system at the same time with their respective local data.

```
1: procedure ParallelSolver(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{Y}, N, d)
           // matrices: \mathbf{V}^{\{j\}} \in \mathbb{R}^{m \times m} for j = 1, ..., d; \mathbf{V}_{\text{up}}, \mathbf{V}_{\text{down}} \in \mathbb{R}^{m \times m}
 2:
           // matrices (cont. 1): \mathbf{Z}_{\mathrm{V,up}}^{\{j\}}, \mathbf{Z}_{\mathrm{V,down}}^{\{j\}} \in \mathbb{R}^{m \times m} for j = 1, ..., d
 3:
           // matrices (cont. 2): \mathbf{Z}_{X,up}, \mathbf{Z}_{X}, \mathbf{Z}_{X,down} \in \mathbb{R}^{m \times k}
 4:
            if my_level == 0 then
 5:
                 // - - - write wings
 6:
                 j_B := 	exttt{my_proc_num(my_level)} \; ; \quad k := N/2
 7:
                 \mathbf{for}\ j=d\ :\ -1\ :\ 1\ \mathbf{do}
 8:
                       if j_B \geq k then
 9:
                             j_B := j_B - k
10:
                        end if
11:
```

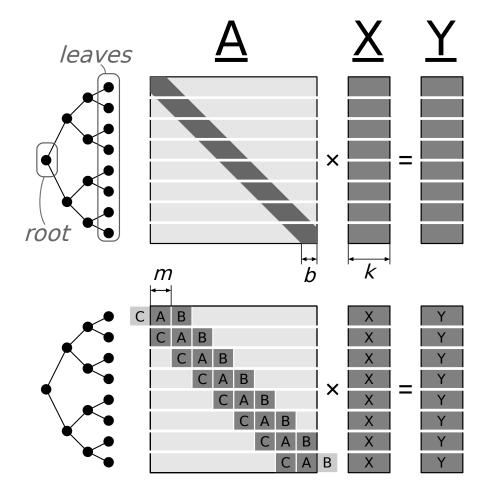


Figure 2: Distributed storage of the linear system. At the top: The linear system of bandwidth  $b \leq m$  is chunked into row-blocks of size m. At the bottom: Each row-block is stored in one leaf using five matrices.

```
\begin{array}{ll} \mbox{if} \ j_B == 0 \ \mbox{then} \\ \mbox{\bf V}^{\{j\}} := \mbox{\bf V}^{\{j\}} + \mbox{\bf B} \ ; \quad \mbox{break for-loop} \end{array}
12:
13:
14:
                                        k := k/2
15:
                             end for
16:
                             j_C := my\_proc\_num(my\_level) - 1;  k := N/2
17:
                             for j = d : -1 : 1 do
18:
                                        if j_C \geq k then
19:
                                                 j_C := j_C - k
20:
                                        end if
21:
                                        if j_C == 0 then
22:
                                                 \mathbf{V}^{\{j\}} := \mathbf{V}^{\{j\}} + \mathbf{C}; break for-loop
23:
                                        end if
24:
                                        k := k/2
25:
                             end for
26:
                              // - - - block-diagonal inversion
27:
                              [\mathbf{V}^{\{1\}},...,\mathbf{V}^{\{d\}},\mathbf{X}]:=\mathbf{A}\backslash[\mathbf{V}^{\{1\}},...,\mathbf{V}^{\{d\}},\mathbf{Y}]
28:
                    end if
29:
                   for \ell = 1 : 1 : d do
30:
                             if my_level == 0 then
31:
                                        \texttt{send\_to\_parent}(\mathbf{V}^{\{\ell\}},...,\mathbf{V}^{\{d\}}\,,\,\mathbf{X})
32:
                             else
33:
                                      receive_from_up_child(\mathbf{V}_{\mathrm{up}}^{\{\ell\}},...,\mathbf{V}_{\mathrm{up}}^{\{d\}},\mathbf{X}_{\mathrm{up}})
receive_from_down_child(\mathbf{V}_{\mathrm{down}}^{\{\ell\}},...,\mathbf{V}_{\mathrm{down}}^{\{d\}},\mathbf{X}_{\mathrm{down}})
34:
35:
                                       if my_level < \ell \ then
36:
                                                 if my_proc_num(\ell-1) is odd then
37:
                                                           \texttt{send\_to\_parent}(\mathbf{V}_{\text{down}}^{\{\ell\}},...,\mathbf{V}_{\text{down}}^{\{d\}}\,,\,\mathbf{X}_{\text{down}})
38:
                                                  else
39:
                                                           \texttt{send\_to\_parent}(\mathbf{V}_{\text{up}}^{\{\ell\}},...,\mathbf{V}_{\text{up}}^{\{d\}}\,,\,\mathbf{X}_{\text{up}})
40:
                                                  end if
41:
                                        end if
42:
43:
                             end if
                              // above: nodes of level \ell receive V_{up}^{\{\ell,...,d\}}, V_{down}^{\{\ell,...,d\}}, X_{up}, X_{down}
44:
                             if my_level == \ell then
45:
                                     \begin{split} \mathbf{S} &:= \begin{bmatrix} \mathbf{I}_{m \times m} & \mathbf{V}_{\text{up}}^{\{\ell\}} \\ \mathbf{V}_{\text{down}}^{\{\ell\}} & \mathbf{I}_{m \times m} \end{bmatrix} \\ & \begin{bmatrix} \mathbf{Z}_{\text{V,up}}^{\{\ell+1\}}, ..., \mathbf{Z}_{\text{V,up}}^{\{d\}} & \mathbf{Z}_{\text{X,up}} \\ \mathbf{Z}_{\text{V,down}}, ..., \mathbf{Z}_{\text{V,down}}^{\{d\}} & \mathbf{Z}_{\text{X,down}} \end{bmatrix} := \mathbf{S} \setminus \begin{bmatrix} \mathbf{V}_{\text{up}}^{\{\ell+1\}}, ..., \mathbf{V}_{\text{up}}^{\{d\}} & \mathbf{X}_{\text{up}} \\ \mathbf{V}_{\text{down}}^{\{\ell+1\}}, ..., \mathbf{V}_{\text{down}}^{\{d\}} & \mathbf{X}_{\text{down}} \end{bmatrix} \\ & \text{send\_to\_up\_child}(\mathbf{Z}_{\text{V,down}}^{\{\ell+1\}}, ..., \mathbf{Z}_{\text{V,down}}^{\{d\}}, \mathbf{Z}_{\text{X,down}}) \\ & \text{send\_to\_down\_child}(\mathbf{Z}_{\text{V,up}}^{\{\ell+1\}}, ..., \mathbf{Z}_{\text{V,up}}^{\{d\}}, \mathbf{Z}_{\text{X,up}}) \\ & \text{send\_to\_down\_child}(\mathbf{Z}_{\text{V,up}}^{\{\ell+1\}}, ..., \mathbf{Z}_{\text{V,up}}^{\{d\}}, \mathbf{Z}_{\text{X,up}}) \end{split}
46:
47:
48:
49:
                             end if
50:
51:
                             if my_level < \ell then
                                       	ext{receive\_from\_parent}(\mathbf{Z}_{	ext{V}}^{\{\ell+1\}},...,\mathbf{Z}_{	ext{V}}^{\{d\}}\,,\,\mathbf{Z}_{	ext{X}})
52:
```

```
if my_level > 0 then
53:
                                              \begin{split} &\texttt{send\_to\_up\_child}(\mathbf{Z}_{V}^{\{\ell+1\}},...,\mathbf{Z}_{V}^{\{d\}}\,,\,\mathbf{Z}_{X}) \\ &\texttt{send\_to\_down\_child}(\mathbf{Z}_{V}^{\{\ell+1\}},...,\mathbf{Z}_{V}^{\{d\}}\,,\,\mathbf{Z}_{X}) \end{split}
54:
55:
                                      end if
56:
                                      if my_level == 0 then
57:
                                               [\mathbf{V}^{\{\ell+1\}},...,\mathbf{V}^{\{d\}}\,,\,\mathbf{X}\,] := [\mathbf{V}^{\{\ell+1\}},...,\mathbf{V}^{\{d\}}\,,\,\mathbf{X}\,] - \mathbf{V}^{\{\ell\}}\cdot[\mathbf{Z}_{\scriptscriptstyle V}^{\{\ell+1\}},...,\mathbf{Z}_{\scriptscriptstyle V}^{\{d\}}\,,\,\mathbf{Z}_{\scriptscriptstyle X}]
58:
                                      end if
59:
                            end if
60:
                   end for
61:
62: end procedure
```

#### 2.1 Derivation of the algorithm

**Origins in SPIKE** We derived the above algorithm by applying the SPIKE algorithm [6] due to Sameh in a divide-and-conquer fashion. We explain the derivation with the help of Figure 3.

Initially, we are given a banded linear system  $\mathbf{A} \cdot \mathbf{X} = \mathbf{Y}$ , as is shown in part 1 of the figure. As is common for the divide-and-conquer approach, the system is split in the middle. We express the system as the composition of equally dimensioned square-matrices  $\mathbf{A}_1, \mathbf{A}_2$ , and matrices  $\mathbf{X}_1, \mathbf{X}_2, \mathbf{Y}_1, \mathbf{Y}_2$ . Since  $\mathbf{A}$  is banded, we need two additional matrices  $\mathbf{V}_1, \mathbf{V}_2$  in order to be able to express  $\mathbf{A}$  as blocks.  $\mathbf{V}_1, \mathbf{V}_2$  are high and thin: They have the hight of half the dimension of the original system and their breadth is equal to the bandwidth of the original system. We give names to  $\tilde{\mathbf{V}}_1, \tilde{\mathbf{V}}_2$  the system in part 2: The system we call blade and the matrices  $\tilde{\mathbf{V}}_1, \tilde{\mathbf{V}}_2$  we call wings. We will come back to this later.

In part 2 of the figure we see a transformed system that is obtained when multiplying the inverse of the block-diagonal matrix  $\mathbf{D}$ 

$$\mathbf{D} = egin{bmatrix} \mathbf{A}_1 & \mathbf{0} \ \mathbf{0} & \mathbf{A}_2 \end{bmatrix}$$

from the left onto the system. If the dimension of  $\mathbf{A}_1$  and  $\mathbf{A}_2$  is large then it is unlikely that  $\mathbf{D}$  is singular. If  $\mathbf{A}$  is symmetric positive definite then  $\mathbf{D}$  is regular with a condition number bounded by that of  $\mathbf{A}$  [10]. Thus, there exist conditions for  $\mathbf{A}$  such that the system considered in part 2 of the figure is well-posed. The obtained system has a interesting structure: It is an identity matrix plus two dense sub-blocks  $\tilde{\mathbf{V}}_1, \tilde{\mathbf{V}}_2$ , that have the breadth of  $\mathbf{A}$ 's bandwidth.

In part 3 of the figure we consider a sub-system that is obtained when extracting the red portions from the matrices in part 2. This sub-system is decoupled from the total system. It has a size to twice the bandwidth and it can be solved directly (e.g. by LU-factorization followed by forward and backward substitutions) for the extracted (red-marked) portion of  $\mathbf{X}$ . This approach would be followed in the usual SPIKE algorithm [8]. However, we use a different approach. As shown in part 3 of our figure, we solve the sub-system with system matrix  $\mathbf{S}$  and write the solution into a matrix that we call  $\mathbf{Z}$ .  $\mathbf{Z}$  is composed vertically of two blocks  $\mathbf{Z}_{\rm up}$  and  $\mathbf{Z}_{\rm down}$ , of which each has the hight of the

bandwidth of A, and the breadth of X and Y.

In part 4 we consider again a modified system. Using  $\mathbf{Z}$  from step 3, we see that we can change the right-hand sides such that the system matrix simplifies to an identity. This transformation can be easily derived: As the first block of the system in part 2, we have

$$\mathbf{I} \cdot \mathbf{X}_1 + [\tilde{\mathbf{V}}_1, \mathbf{0}] \cdot \mathbf{X}_2 = \tilde{\mathbf{Y}}_1$$
.

Replacing  $[\tilde{\mathbf{V}}_1, \mathbf{0}] \cdot \mathbf{X}_2$  by  $\tilde{\mathbf{V}}_1 \cdot \mathbf{Z}_{\text{down}}$  and moving the second term to the right-hand side yields the formula for  $\hat{\mathbf{Y}}_1$ . The formula for  $\hat{\mathbf{Y}}_2$  can be found in an analogous way.

**Parallelism, recursions, and the computing system** In Figure 3, there are two stages where parallelism can be exploited: The transformation from part 1 to part 2 involves the solution of the following banded linear systems:

$$\mathbf{A}_1 \cdot [\tilde{\mathbf{V}}_1, \tilde{\mathbf{Y}}_1] = [\mathbf{V}_1, \mathbf{Y}_1] \tag{2a}$$

$$\mathbf{A}_2 \cdot [\tilde{\mathbf{V}}_2, \tilde{\mathbf{Y}}_2] = [\mathbf{V}_2, \mathbf{Y}_2] \tag{2b}$$

These two systems can be solved independently from each other. The second stage, where parallelism can be exploited, is in the computation of the two components  $\hat{\mathbf{Y}}_1$  and  $\hat{\mathbf{Y}}_2$ :

$$\hat{\mathbf{Y}}_1 = \tilde{\mathbf{V}}_1 \cdot \mathbf{Z}_{\text{down}} \tag{3a}$$

$$\hat{\mathbf{Y}}_2 = \tilde{\mathbf{V}}_2 \cdot \mathbf{Z}_{\text{up}} \tag{3b}$$

The problems in (2) and (3) can be expressed as recursions. The recursion for (2) is obvious because we develop an algorithm that solves banded linear system and interiorly requires the solution of smaller banded linear systems. A recursion for (3) is found by distributing the computation over the rows of  $\hat{\mathbf{Y}}_1$ ,  $\hat{\mathbf{Y}}_2$ . This is shown in Figure 4. The computation of each  $\hat{\mathbf{Y}}_1$  and  $\hat{\mathbf{Y}}_2$  can each be expressed in a form as  $\mathbf{M} := \mathbf{M} - \mathbf{U} \cdot \mathbf{W}$ , in some programming languages also written as  $\mathbf{M} = \mathbf{U} \cdot \mathbf{W}$ . The figure shows how the computation of the update of  $\mathbf{M}$  can be distributed through a two-tree network by dividing it in vertical direction. All-together, we can formulate our whole algorithm for solving a banded linear system in a recursive way. The following code demonstrates this:

```
1: procedure RecursiveSolver(A, Y)
         if (thenA has small dimension)
2:
3:
              \mathbf{X} = \mathbf{A} \backslash \mathbf{Y}
              return X
4:
         end if
5:
         Decompose the system into A_1, A_2, V_1, V_2, Y_1, Y_2, X_1, X_2.
6:
         [\tilde{\mathbf{V}}_1, \tilde{\mathbf{Y}}_1] := \text{RecursiveSolver}(\mathbf{A}_1, [\mathbf{V}_1, \mathbf{Y}_1])
7:
         [\mathbf{V}_2, \mathbf{Y}_2] := \text{RecursiveSolver}(\mathbf{A}_2, [\mathbf{V}_2, \mathbf{Y}_2])
8:
         Compose S and solve the reduced linear system for \mathbf{Z}_{up}, \mathbf{Z}_{down}.
9:
```

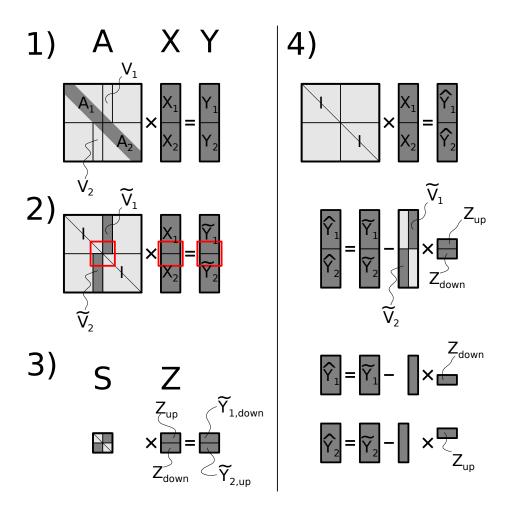


Figure 3: Derivation of our algorithm by applying SPIKE in a divide-and-conquer fashion. The system is divided into two equally sized systems of smaller dimension. After solving these, the solutions can be put together through the solution of a decoupled system (part 3) that has a small dimension.

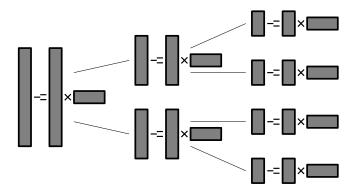


Figure 4: Recursive divide-and-conquer approach to compute the matrix update  $\mathbf{M} - = \mathbf{U} \cdot \mathbf{W}$  on a parallel two-tree computing system. Each node divides the computational problem vertically into two problems of smaller dimension. These are then solved recursively by its children.

```
10: // \hat{\mathbf{Y}}_1 <u>is</u> \tilde{\mathbf{Y}}_1, \hat{\mathbf{Y}}_2 <u>is</u> \tilde{\mathbf{Y}}_2

11: \hat{\mathbf{Y}}_1 := \text{RECURSIVEMATMUL}(\tilde{\mathbf{Y}}_1, \tilde{\mathbf{V}}_1, \mathbf{Z}_{\text{down}})

12: \hat{\mathbf{Y}}_2 := \text{RECURSIVEMATMUL}(\tilde{\mathbf{Y}}_2, \tilde{\mathbf{V}}_2, \mathbf{Z}_{\text{up}})

13: // \mathbf{X} <u>is</u> \begin{bmatrix} \hat{\mathbf{Y}}_1 \\ \hat{\mathbf{Y}}_2 \end{bmatrix}

14: return \mathbf{X}

15: end procedure
```

The computing system has been chosen as a two-tree in order to exploit the recursive nature of the algorithm: Initially, the root has the solve the linear system. According to the above code it would call its children in lines 7–8 to solve recursively the subsystems with  $\mathbf{A}_1$  and  $\mathbf{A}_2$ . This is supposed to be done in parallel. Afterwards, the root composes  $\mathbf{S}$  from the small portions  $\tilde{\mathbf{V}}_{1,\text{down}}, \tilde{\mathbf{V}}_{2,\text{up}}$  of  $\tilde{\mathbf{V}}_1, \tilde{\mathbf{V}}_2$ , and computes  $\mathbf{Z}_{\text{up}}, \mathbf{Z}_{\text{down}}$ . Finally, it calls again its children in order to compute  $\hat{\mathbf{Y}}_1, \hat{\mathbf{Y}}_2$ .

Access pattern on the distributed memory So far our explanation helps to understand how the algorithm works and why the results for  $\mathbf{X}$  are correct. But yet it is not easy to see how communication-intense the algorithm is and how the recursion acts on the global system. In this paragraph we give illustrations for both.

Our algorithm can be interpreted in an elegant way. To solve a system

$$\mathbf{A} \cdot \mathbf{X} \cdot \mathbf{Y}$$

we can interprete that the algorithm applies an iterative scheme

$$\begin{split} \mathbf{A}^{(0)} &:= \underline{\mathbf{A}} & \mathbf{Y}^{(0)} := \underline{\mathbf{Y}} \\ \mathbf{A}^{(j+1)} &:= \left(\mathbf{D}^{(j)}\right)^{-1} \cdot \mathbf{A}^{(j)} & \mathbf{Y}^{(j+1)} := \left(\mathbf{D}^{(j)}\right)^{-1} \cdot \mathbf{Y}^{(j)} & \text{for } j = 0, ..., d \,, \end{split}$$

where  $d = \log_2(N)$  and where  $\mathbf{A}^{(d+1)} = \mathbf{I}$  and thus  $\underline{\mathbf{X}} = \mathbf{Y}^{(d+1)}$ . The matrices  $\mathbf{D}^{(j)}$  are block-diagonal matrices whose blocks are blades. As we have seen, the inverses of blades can be computed in parallel through RecursiveMatMul. We only need one upwards-communication of  $\mathbf{V}_{1,\mathrm{up}}$ ,  $\mathbf{V}_{2,\mathrm{down}}$  and one downwards communication of  $\mathbf{Z}_{\mathrm{down}}$ ,  $\mathbf{Z}_{\mathrm{up}}$ .

Figure 5 illustrates the iteration for N=8, d=3. The figure shows the system matrix in the right part, starting at the top with the original matrix and ending at the bottom with an identity. The left part of the figure shows the matrices  $\mathbf{D}^{(j)}$  for j=0,...,d.

We want to explain how this interpretation of the algorithm's action can be derived from RecursiveSolver and at the same time discuss the algorithmic steps as represented by the figure. The recursion in lines 7–8 results in a partitioning of the system matrix along the diagonal blocks. Black lines indicate the recursive diagonal partitioning. Grey coloring shows the non-zero pattern. In the bottom of the recursion the inverses of the smallest diagonal blocks are applied from the left onto the system. So the matrix  $\mathbf{D}^{(0)}$  consists solely of diagonal elements. The matrix  $A^{(1)}$  has identity matrices on the diagonal blocks by construction. The sub-diagonal blocks are no longer triangular but dense. Since level 0 is the recursive bottom, we now ascend. The matrix  $\mathbf{D}^{(1)}$  consists of four blades. Multiplication from the left with its inverse yields a matrix  $A^{(2)}$  that has identities on larger diagonal blocks (since these where identical to the blades on the diagonals of  $\mathbf{D}^{(1)}$ ). However, the hight of the wings in  $\mathbf{A}^{(2)}$  increases because there are subdiagonal blocks in  $\mathbf{A}^{(1)}$  that were not represented in  $\mathbf{D}^{(1)}$ . Further ascending, the matrix  $\mathbf{D}^{(2)}$  consists of two diagonal blocks, which are blades of dimension 4. The two subdiagonal blocks in  $A^{(3)}$  suffer from fill-in in vertical direction while the diagonal blocks become identity matrices. Very finally,  $\mathbf{A}^{(3)}$  has a blade-structure and this  $(\mathbf{D}^{(3)})^{-1} \cdot \mathbf{A}^{(3)}$  yields the identity.

From the sparsity patterns of the system matrix in each iteration we can draw conclusions on the memory that the nodes need to hold in order to be able to compute  $\mathbf{A}^{(0)},...,\mathbf{A}^{(d+1)}$  without the need of any overhead for, e.g., dynamically changing a sparse-memory representation of  $\mathbf{A}$ . In our algorithm ParallelSolver we store  $\underline{\mathbf{A}}$  as wing-matrices  $\mathbf{V}^{\{j\}}$ , j=1,...,d. The top of Figure 6 depicts this: The fill-in pattern of  $\underline{\mathbf{A}}$  fits into  $\log_2(N)$  column vectors.

For this data structure the product of  $\underline{\mathbf{A}}$  with an inverse of on of the above block-diagonal matrices  $\mathbf{D}^{(j)}$  can be computed efficiently, as is shown in the bottom of the figure: Say we want to compute the product with  $(\mathbf{D}^{(j)})^{-1}$  for j=3. In this case, the leaves send the data for the reduced system (compare to the red-framed portions in Figure 3 part 2) to the root of the sub-trees of level j=3. The roots of the subtrees compute the reduced solutions  $\mathbf{Z}_{\rm up}, \mathbf{Z}_{\rm down}$ , that afterwards they send back to all leaves through their sub-tree. The leaves update the wing-matrices by performing the same computations for them as for updating  $\mathbf{Y}$ .

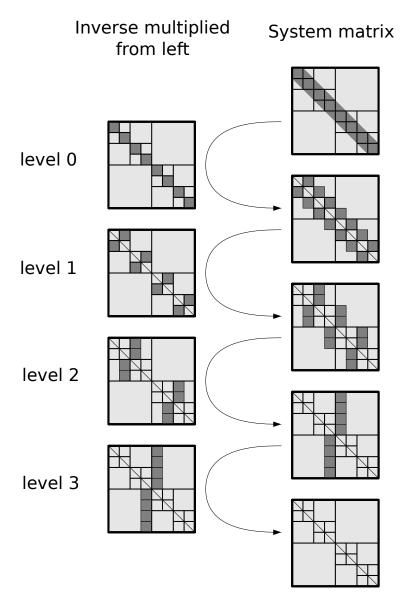
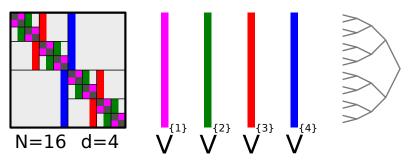


Figure 5: Successive multiplication of inverse of block-diagonal matrix from the left onto the system matrix. The block-diagonal matrices consist of blades. Their inverse is easy to apply in parallel.

# Data structure for system matrix



## Product with inverse of blade on (sub-)two-tree

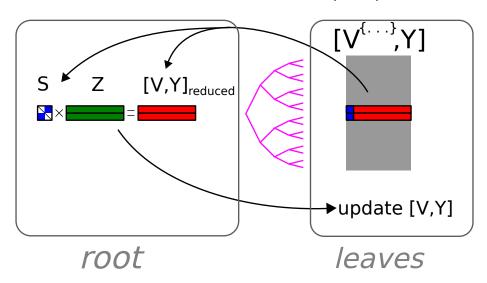


Figure 6: Top: The matrix is represented by  $d = \log_2(N)$  dense wing-matrices  $\mathbf{V}^{\{j\}}, \ j = 1,...,d$ , which are each distributed in row-chunks over the leaves. Bottom: The product with the inverse of a blade. The leaves send extracted matrices to the root of the sub-tree. The root computes the decoupled solution  $\mathbf{Z}$  and sends it back through the tree to all leaves, so they can update the wing-matrices.

**Parallel time complexity** We derive the time complexity by the following list of axioms:

- (i) According to Figure 5 the algorithm performs  $\mathcal{O}(\log(N))$  iterations.
- (ii) The communicated pieces of data per iteration are the node-local matrices

$$[\mathbf{V}^{\{1\}},...,\mathbf{V}^{\{d\}},\mathbf{Y}] \in \mathbb{R}^{m \times (m \cdot d + k)}$$

which consist of  $\mathcal{O}(m \cdot (m \cdot \log(N) + k))$  elements. The communication cost per iteration is bounded by the time required for sending these elements from an arbitrary leaf to the root or vice versa (size  $\mathbf{Z}_{V}, \mathbf{Z}_{X}$  have the same dimensions as  $[\mathbf{V}^{\{1\}}, ..., \mathbf{V}^{\{d\}}, \mathbf{Y}]$ ). Equivalently, the time complexity for communication per iterations is

$$\mathcal{O}(c_N^{\text{Lat}} + m \cdot (m \cdot \log(N) + k)),$$

where  $c_N^{\text{Lat}}$  is the time complexity of physical time that is needed to send a single number from the root to a leaf that is farthest away from the root in terms of cable-length.

(iii) The computational complexity per iteration per node is as follows: The nodes that are not leaves do either compute nothing or they compute a decomposition of a matrix  $\mathbf{S} \in \mathbb{R}^{(2 \cdot m) \times (2 \cdot m)}$  and apply it to the columns of the matrix  $[\mathbf{V}^{\{1\}}, ..., \mathbf{V}^{\{d\}}, \mathbf{Y}]$  in order to compute  $\mathbf{Z}_{V}, \mathbf{Z}_{X}$ . The leaves on the other hand compute matrix-matrix-products of an  $m \times m$ -matrix with  $\mathbf{Z}_{V}$  and  $\mathbf{Z}_{X}$ . Thus, the computational complexity per node per iteration is bounded by  $\mathcal{O}(m^{3} \cdot \log(N) + m^{2} \cdot k)$ .

Combining the items, we find that the parallel time complexity of our method is:

$$\mathcal{O}\left(\log(N)\cdot\left(c_N^{\text{Lat}} + m^2\cdot\left(m\cdot\log(N) + k\right)\right)\right)$$
 (4)

Unfortunately, for a computing system of N nodes with each of a size in  $\Theta(1)$  the minimum possible value for  $c_N^{\text{Lat}}$  lives in  $\Theta(N^{1/3})$ .

We consider two special cases:

1. Assuming  $k, m \in \mathcal{O}(1)$  the complexity result simplifies to

$$\mathcal{O}\Big(\log(N)\cdot N^{1/3}\Big)$$
.

2. Assuming  $k, m \in \Omega(N^{1/6})$ , the time complexity is

$$\mathcal{O}\left(\log(N) \cdot m^2 \cdot \left(m \cdot \log(N) + k\right)\right).$$

Whereas the second result is obviously optimal for solving a band-matrix with dense band of bandwidth m (since it is has N diagonal blocks of size m that need to be factorized), it turns out that the first result is not optimal and can be made optimal by a fine-tuning.

# 3 Lower complexity bound for the parallel solution of tridiagonal linear systems

In this section we prove in order the following statements, where s is the physical dimension in which the computing system is built (e.g., if the computing system is built on the surface of a planet then s=2, and if the computing system is a/the planet then s=3):

- 1. The latency  $c_N^{\rm Lat}$  is bounded from below by  $\Omega(N^{1/s})$ , regardless how many cables are used.
- 2. The time complexity for an algorithm for solving the tridiagonal linear equation system of dimension N on a distributed memory machine of computing nodes is bounded from below by  $\Theta(N^{1/(s+1)})$ , no matter how many nodes and cables are used.

**Latency** We discuss on the latency time for a computing system of N nodes that are s-dimensional spheres, where the latency of one node to communicate to another is bounded from below by the order of their physical distance in space.

For N nodes, the diameter on a line is  $\geq N-2$  because the two outermost nodes have  $N-2\in\Omega(N)$  nodes between themselves. For an illustration of this, consider Figure 2, that shows twice the topology for the two-tree computing system of N=8 nodes. Since the leaves are placed on a line with a unit distance, the physical distance of the root to the farthest leaf is bounded from below in  $\Omega(N)$ .

Now let us consider the case where we use s=2 dimensions: We place the computing nodes into the smallest possible circle. Since the total area of N nodes is  $\Theta(N)$ , the radius r of the must be in  $r \in \Theta(N^{1/2})$ . Then, the communication between two nodes requires at most a time of  $2 \cdot r \in \Theta(N^{1/2})$ . In s=3 dimensions the situation is even better. Here the radius must only be of length  $r \in \Theta(N^{1/3})$ . Given the positions of the nodes in the sphere, it is trivial to find an almost optimal communication network for them: Connecting them as a tree yields that the maximum diameter of the communication tree is  $\mathcal{O}(\log(N) \cdot N^{1/s})$  because each cable can at most have length  $\mathcal{O}(N^{1/s})$  and a tree network has diameter  $\mathcal{O}(\log(N))$ .

Lower bound on time complexity for solving the tridiagonal linear systems. Presume we shall solve a tridiagonal system  $\mathbf{A} \cdot \mathbf{x} = \mathbf{y}$  of dimension N. Presume that we use  $\mathcal{O}(N^p)$  nodes of which each holds at most  $\mathcal{O}(N^q)$  pieces of data. It must be  $q \geq 1-p$  because  $\mathcal{O}(N)$  pieces of data must be stored in total.

It is known that each number of the solution vector  $\mathbf{x}$  depends on each number of the right-hand side  $\mathbf{y}$  and each value of the matrix  $\mathbf{A}$ . Thus the following hold:

dimension $s$	order $p$ of number of nodes	order $\omega$ of time complexity
1	1/2	1/2
2	2/3	1/3
3	3/4	1/4

Table 1: Lower complexity bounds for solving a linear equation system on parallel a distributed memory system of arbitrary many nodes.

- (i) Each node must read at least  $\mathcal{O}(N^{1-p})$  pieces of its data.
- (ii) Each node must communicate at least once (either directly or indirectly) to each other node.

Combining the two properties, we find a lower bound for the time complexity with a fixed value of p:

$$\Omega(\underbrace{N^{1-p}}_{\text{read problem}} + \underbrace{N^{\frac{p}{s}}}_{\text{communicate}}) = \mathcal{O}(N^{\omega})$$

The minimum of polynomial complexity orders  $\omega$  depending on s and the optimal order p of the number of nodes are given in table 1.

The question arises why we have not presented our algorithm with a number of nodes that is in  $\mathcal{O}(N^{3/4})$  since then our algorithm would be logarithmically close to the optimal time complexity. We did not because for our PhD thesis we will have to solve problems where  $N \approx 10^6$  and  $m \approx 100$ . So for our applications the execution time is rather dominated by  $\log(N)^2 \cdot m^3$ . The  $m^3$  arises from the fact that  $\underline{\mathbf{A}}$  has N dense  $m \times m$ -matrices on the diagonal that must be decomposed. In theory a fast matrix-multiplication algorithm could be employed to reduce the complexity order for this, but this is not practical. Thus, for the problems that we need to solve, the complexity of our algorithm is already logarithmically close to optimal or maybe even optimal.

## 4 Conclusions

We presented an algorithm for the efficient parallel solution of (block-)tridiagonal linear systems. We provided an accurate implementation of the algorithm that uses a common message-passing interface. Following our analysis, the algorithm has a parallel time complexity that could be made optimal for tridiagonal systems (by simply using fewer nodes) and that is clearly optimal for block-tridiagonal linear systems, respectively banded linear systems of small bandwidth.

Though a proper implementation has been provided for the algorithm, it will be rather difficult to utilize its full potential in practice. This is because the time that is required to send the data to the solver would already destroy the benefit. Software that uses this solver needs to be highly sophisticated. In particular, problems must be instantiated in a way such that the linear system

is already distributed in the memory of the leaves of our computing system at the time when our solver is called.

Further work will be related to an attempt of implementing an optimal control solver by direct transcription that shall solve the linear systems within the non-linear programming solver by means of the linear system solver that has been presented in this work.

### References

- [1] P. Amodio, L. Brugnanot, and J. Plemmons. Parallel factorizations and parallel solvers for tridiagonal linear systems. *Linear Algebra Appl*, pages 347–364, 1992.
- [2] Travis Austin, Markus Berndt, and David Moulton. A memory efficient parallel tridiagonal solver. Preprint LA-UR-03-4149, 2004.
- [3] Stefan Bondeli. Paper: Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations. *Parallel Comput.*, 17(4-5):419–434, July 1991.
- [4] Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening multigrid on distributed memory machines. SIAM J. Scientific Computing, 21(5):1823–1834, 2000.
- [5] R. W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *J. ACM*, 12(1):95–113, January 1965.
- [6] Murat Manguoglu, Ahmed H. Sameh, and Olaf Schenk. Pspike: A parallel hybrid sparse linear system solver. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, Euro-Par 2009 Parallel Processing, pages 797–808, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [7] Nathan Mattor, Timothy J. Williams, and Dennis W. Hewett. Algorithm for solving tridiagonal matrix problems in parallel. *Parallel Computing*, 21(11):1769 1782, 1995.
- [8] Eric Polizzi and Ahmed Sameh. Spike: A parallel environment for solving banded linear systems. *Computers and Fluids*, 36(1):113–120, 2007.
- [9] R. D. Russell and L. F. Shampine. A collocation method for boundary value problems. *Numerische Mathematik*, 19(1):1–28, Feb 1972.
- [10] Y. Saad. Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.
- [11] Harold S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM*, 20(1):27–38, January 1973.

- [12] H. H. Wang. A parallel method for tridiagonal equations. ACM Trans. Math. Softw., 7(2):170-183, June 1981.
- [13] Plamen Yalamov and Velisar Pavlov. On the stability of a partitioning algorithm for tridiagonal systems. SIAM J. Matrix Anal. Appl., 20(1):159-181, January 1999.