Multi-threaded Nested Filtering Factorization Preconditioner

Pawan Kumar⋆, Karl Meerbergen, and Dirk Roose

Department of Computer Science,
KU Leuven, Celestijnenlaan 200A, 3001 Heverlee-Leuven, Belgium
{pawan.kumar,karl.meerbergen,dirk.roose}@cs.kuleuven.be

Abstract. The scalability and robustness of a class of non-overlapping domain decomposition preconditioners using 2-way nested dissection reordering is studied. In particular, three methods are considered: a nested symmetric successive over-relaxation (NSSOR), a nested version of modified ILU with rowsum constraint (NMILUR), and nested filtering factorization (NFF). The NMILUR preconditioner satisfies the rowsum property i.e., a right filtering condition on the vector $(1, \ldots, 1)^T$. The NFF method is more general in the sense that it satisfies right filtering condition on any given vector. There is a subtle difference between NMILUR and NFF, but NFF is much more robust and converges faster than NSSOR and NMILUR. The test cases consist of a Poisson problem and convection-diffusion problems with jumping coefficients.

1   Introduction

We consider the problem of solving large sparse linear systems of the form

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

by an iterative method preconditioned by a nonoverlapping domain decomposition method. A preconditioner $\mathbf{B}$ is said to satisfy "rowsum" property when

$$\mathbf{B1} = \mathbf{A1} \tag{2}$$

where $\mathbf{1} = [1, 1, 1, ..., 1]^T$. In general, a preconditioner may satisfy a more general (right) filtering condition as follows

$$\mathbf{Bt} = \mathbf{At} \tag{3}$$

where $\mathbf{t}$ is a filter vector. The basic linear fixed point method for solving the linear system (1) above is given as follows

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \mathbf{B}^{-1}(\mathbf{b} - \mathbf{Ax}^n) = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{x}^n + \mathbf{B}^{-1}\mathbf{b}. \tag{4}$$

⋆ This work was done when the first author had a visiting position at Université Libre de Bruxelles, Brussels and a postdoctoral position at KU Leuven, Leuven and Flanders Exascience Lab (Intel labs Europe), Leuven.

Now, subtracting (4) from the identity $\mathbf{x} = \mathbf{x} - \mathbf{B}^{-1}\mathbf{A}\mathbf{x} + \mathbf{B}^{-1}\mathbf{b}$, we obtain the following expression of the error at the $(n+1)$th step

$$\mathbf{e}^{n+1} = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{e}^n = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})^2\mathbf{e}^{n-1} = \cdots = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})^{n+1}\mathbf{e}^0.$$

If $\mathbf{B}$ satisfies the filtering condition (3), we have $(\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{t} = \mathbf{0}$, thus by choosing a suitable $\mathbf{t}$, a desired component of the error vector could be removed. Good candidates for the filter vectors are approximations to the eigenvectors corresponding to the smallest eigenvalues of the preconditioned coefficient matrix: $\mathbf{B}^{-1}\mathbf{A}$ [17].

In the past, several approximate and exact filtering preconditioners were proposed for block tridiagonal matrices of the form $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. The matrix $\mathbf{A}$ is partitioned as follows

$$\mathbf{A} = \begin{pmatrix} D_1 & U_1 & & \\ L_1 & D_2 & \ddots & \\ & \ddots & \ddots & U_{n-1} \\ & & L_{n-1} & D_n \end{pmatrix}, \tag{5}$$

where the matrices $\mathbf{L}$, $\mathbf{D}$, and $\mathbf{U}$ are sparse matrices as follows

$$\mathbf{L} = \begin{pmatrix} 0 & & & \\ L_1 & 0 & & \\ & \ddots & \ddots & \\ & & L_{n-1} & 0 \end{pmatrix}, \quad \mathbf{D} = \begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_n \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 0 & U_1 & & \\ & \ddots & \ddots & \\ & & 0 & U_{n-1} \\ & & & 0 \end{pmatrix}.$$

The exact block $\mathbf{LU}$ factorization of the matrix $\mathbf{A}$ reads

$$\mathbf{A} = (\mathbf{T} + \mathbf{L})(\mathbf{I} + \mathbf{T}^{-1}\mathbf{U}),$$

where $\mathbf{T}$ is a block diagonal matrix obtained from the following recurrence

$$T_i = \begin{cases} D_1, & i = 1, \\ D_i - L_{i-1}T_{i-1}^{-1}U_{i-1}, & 1 < i \le n. \end{cases} \tag{6}$$

The Schur complements T_i, $(i > 1)$ are costly to compute because the sub-matrices $T_i, i > 1$ usually become denser (even though T_1 is sparse). There exist many preconditioners that involve some approximation of the Schur complement. However, we are particularly interested in the approximations that retain the filtering condition (3). For some approximations, satisfying filtering condition with vector $\mathbf{1}$ is particularly useful for convection-diffusion problems as we will see later. One of the earliest known methods satisfying (3) is the modified incomplete LU [17]; this method is applicable to general matrices that may not have block tridiagonal form. For block tridiagonal matrices, in [18], a tridiagonal approximation to T_{i-1}^{-1} and $L_{i-1}T_{i-1}^{-1}U_{i-1}$ was proposed. Axelson and Polman [16] proposed an approximation that satisfies the following relation

$$\tilde{T}_i\mathbf{1} = (D_i - L_{i-1}\tilde{T}_{i-1}^{-1}U_{i-1})\mathbf{1},$$

$$\tilde{T}_i\mathbf{n} = (D_i - L_{i-1}\tilde{T}_{i-1}^{-1}U_{i-1})\mathbf{n},$$

where $\mathbf{1}$ and $\mathbf{n} = [1, 2, \ldots, n]^T$ are test vectors. In [15], a sequence of filtering decompositions is proposed where the choice of filter vectors are the sine and cosine functions that damp both the high and low frequency components of the error. Recently, improved filtering decompositions were proposed [13, 14]. A parallel implementation of a filtering decomposition appeared in [12].

In [9] a class of parallel preconditioners based on a 2-way nested dissection (ND) ordering was proposed. In particular, a method named nested filtering factorization (NFF) was proposed. Given a matrix $\mathbf{A}$, the 2-way ND reordering leads to a permuted matrix $\mathbf{P^T AP}$ with the following structure

$$\mathbf{P^T AP} = \begin{pmatrix} D_0 & & U_0 \\ & D_1 & U_1 \\ \hline L_0 & L_1 & S_0 \end{pmatrix} = (\mathbf{T} + \mathbf{L})(\mathbf{I} + \mathbf{T}^{-1}\mathbf{U}), \tag{7}$$

where

$$\mathbf{L} = \begin{pmatrix} & & \\ \hline L_0 & L_1 & \end{pmatrix}, \quad \mathbf{T} = \begin{pmatrix} T_0 & & \\ & T_1 & \\ \hline & & T_2 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} & & U_0 \\ & & U_1 \\ \hline & & \end{pmatrix}.$$

This leads to the following recursion for T_i

$$T_i = \begin{cases} D_i, & i = 0, 1, \\ S_0 - L_0 T_0^{-1}U_0 - L_1 T_1^{-1}U_1, & i = 2. \end{cases}$$

Unlike (6), where T_i depends on T_{i-1}, for the recursion above, T_0 and T_1 are independent but T_2 depends on both T_0 and T_1. To achieve more concurrency in the algorithm, we apply the 2-way ND reordering to the domains D_0 and D_1 and obtain a similar factorization. The factored forms of D_0 and D_1 could then be used to estimate T_2. The preconditioners are obtained by approximating the Schur complements as was done in the block tridiagonal case. In this paper, we consider three possible approximations (first introduced in [11]) as follows:

- **NSSOR**: Here $T_2 = S_0$, the terms $L_0 T_0^{-1}U_0$ and $L_1 T_1^{-1}U_1$ are dropped. We call this NSSOR because the method is a multilevel extension of the classical SSOR method.
- **NMILUR**: Here T_2 is approximated as follows

$$T_2 = S_0 - \text{diag}(L_0 T_0^{-1}U_0\mathbf{1}) - \text{diag}(L_1 T_1^{-1}U_1\mathbf{1}).$$

- **NFF**: Here T_2 is approximated as follows

$$T_2 = S_0 - L_0\beta_1 U_0 - L_1\beta_2 U_1$$

  where

$$\beta_1 = \text{diag}(T_0^{-1}(U_0 t_0)./(L_0 t_0)), \ \beta_2 = \text{diag}(T_1^{-1}(U_1 t_1)./(L_1 t_1)),$$

  here ./ is a point-wise vector division, diag is the MATLAB operator, t_0 and t_1 are given column vectors.

Table 1. Number of iterations for the Poisson problem on $40 \times 40 \times 40$ grid with zero boundary condition. Preconditioners used are AMG: BoomerAMG in Hypre, AGMG: unsmoothed aggregation based multigrid [19], CG: conjugate gradient, PARASAILS: sparse approximate preconditioner in Hypre [2], tolerance for the relative residual is 10^{-7}. Ndoms stands for number of subdomains.

| Solvers	GMRES			CG			
Preconditioners/Ndoms	NSSOR	NMILUR	NFF	AMG	AGMG	PARASAILS	none
2	24	60	12	5	12	52	92
4	27	86	13	6	12	55	92

Notice the subtle difference in the approximations for NMILUR and NFF both satisfying a right filtering property; NMILUR satisfies right filtering on the vector of all ones and NFF satisfies filtering on any given filter vector **t**. As we will see later, NFF is remarkably fast and robust compared to NMILUR and NSSOR and is a "potential" competitor of the algebraic multigrid methods (AMG). In Table 1, we compare the iteration count of GMRES preconditioned by NSSOR, NMILUR and NFF, and CG preconditioned by state-of-the-art preconditoners of Hypre [2] (PARASAILS and boomer AMG) and AGMG (aggregation based AMG) [19]. Unlike AMG methods which rely on smoothing and a coarse grid solve, the methods considered in this paper belong to a class of multilevel Schur complement based methods. In this paper, we present preliminary results on the scalability of these methods on a shared memory architecture. Our choice of using 2-way ND is justified as follows:

- The recursive 2-way ND reordering leads to a blindly cache-aware algorithm exhibiting both spatial and temporal locality as illustrated in Figure 3.
- The matrix-vector product, setup, and solution phases (forward and backward solves) can be executed concurrently as explained in the sections that follow.
- It is well known that ND is a fill-reducing reordering for direct methods. Such reorderings also reduce fill-ins for the preconditioners considered in this paper.

We show a straightforward parallelization of the methods using `cilk plus` with new vectorization features using elemental functions. We also identify the possible future work that may lead to better parallelization. Our numerical experiments consist of a model Poisson problem and convection-diffusion problem with jumping coefficients.

2 Nested Filtering Factorization

As mentioned above, effective parallelism and cache reuse can be achieved by the 2-way nested dissection reordering first proposed by Alan George [10] in 1973 for a regular finite element mesh. The method has been extended to tackle general problems where only the adjacency graph of the matrix is required [6–8].
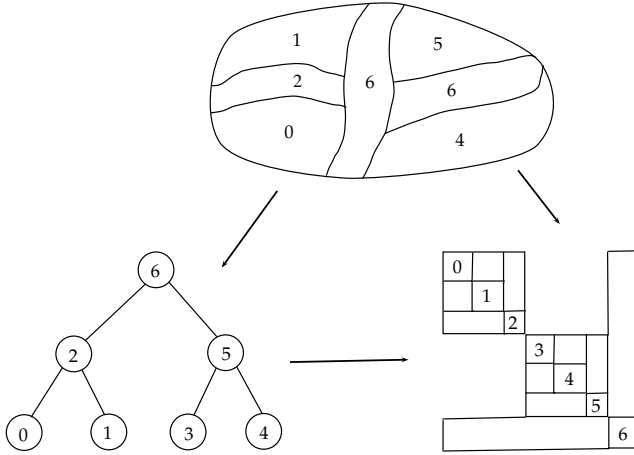
Fig. 1. Top: Graph partitioned into 4 parts, the four parts are first numbered, followed by the separator. Bottom Left: Corresponding tree, Bottom right: Corresponding matrix after reordering.

In Fig. 1, we illustrate the ND reordering, the corresponding tree graph, and the structure of the ND reordered matrix. In the 2-way ND method, a separator (vertices or edges of the graph) is selected which (when removed) separates the graph into two or more subgraphs. The process is continued recusively on the separated subgraphs. In Fig. 1, we show the typical block structure of a ND reordered matrix; such structure is obtained by renumbering the nodes in the separated graphs first, followed by numbering the nodes in the separator. In Fig. 1 we show a graph partitioned into four parts. Renumbering the nodes is equivalent to permuting the matrix. Assume that ND has been applied recursively $l + 1$ times. We will refer to the top level as level $l + 1$ (vertex 6 of Fig. 1 is at top level) and the bottom level as level 0 (vertices 0, 1, 3, and 4 in Fig. 1). Let $\mathbf{P} \in \mathbb{N}^{n \times n}$ be the required permutation matrix, then the permuted matrix corresponding to ND reordering is a bordered block diagonal matrix with the following structure

$$\mathbf{P^T A P} = \begin{pmatrix} D_0^l & & U_0^l \\ & D_1^l & U_1^l \\ \hline L_0^l & L_1^l & S_0^{l+1} \end{pmatrix} = (\mathbf{T_l} + \mathbf{L_l})\mathbf{T_l}^{-1}(\mathbf{T_l} + \mathbf{U_l}), \tag{8}$$

where

$$\mathbf{L_l} = \begin{pmatrix} & & \\ \hline L_0^l & L_1^l & \end{pmatrix}, \quad \mathbf{T_l} = \begin{pmatrix} D_0^l & & \\ & D_1^l & \\ \hline & & T_0^{l+1} \end{pmatrix}, \quad \mathbf{U_l} = \begin{pmatrix} & & U_0^l \\ & & U_1^l \\ \hline & & \end{pmatrix}.$$

From (8), it is easy to see that T_0^{l+1} is

$$T_0^{l+1} = S_0^{l+1} - L_0^l(D_0^l)^{-1}U_0^l - L_1^l(D_1^l)^{-1}U_1^l. \tag{9}$$

Notice that T_0^{l+1} depends on D_0^l and D_1^l. Both D_0^l and D_1^l themselves have nested bordered block diagonal structure as follows

$$D_0^l = \begin{pmatrix} D_0^{l-1} & & U_0^{l-1} \\ & D_1^{l-1} & U_1^{l-1} \\ \hline L_0^{l-1} & L_1^{l-1} & S_0^l \end{pmatrix}, \quad D_1^l = \begin{pmatrix} D_2^{l-1} & & U_2^{l-1} \\ & D_3^{l-1} & U_3^{l-1} \\ \hline L_2^{l-1} & L_3^{l-1} & S_1^l \end{pmatrix},$$

and both of them admit a block **LU** factorization as we had for the original matrix $\mathbf{P^T A P}$ in (8). The factored forms of D_0^l and D_1^l could then be used to estimate (9). This process of estimating the Schur complement by using the block **LU** factorization of the subdomain matrices is continued till the second last level (the last level 0 does not have ND structure). At the second last level $l = 1$, the domain matrices are represented by D_k^1, $k = 0, \ldots, 2^l - 1$ where

$$D_k^1 = \begin{pmatrix} D_{2k}^0 & & U_{2k}^0 \\ & D_{2k+1}^0 & U_{2k+1}^0 \\ \hline L_{2k}^0 & L_{2k+1}^0 & S_k^1 \end{pmatrix}, \quad k = 0, \ldots, 2^l - 1. \quad (10)$$

As above, to obtain a block **LU** factorization for D_k^1, $k = 0, \ldots, 2^l - 1$, we need a Schur complement computation

$$T_k^1 = S_k^1 - L_{2k}^0 (D_{2k}^0)^{-1} U_{2k}^0 - L_{2k+1}^0 (D_{2k+1}^0)^{-1} U_{2k+1}^0, \quad k = 0, \ldots, 2^l - 1, \quad (11)$$

where D_{2k}^0 and D_{2k+1}^0 are assumed to be small enough to be factored cheaply and easily by a direct method. If the Schur complements T_k^j are not approximated, we obtain an exact nested factorization, but our objective here is to obtain preconditoners by avoiding the exact computation of the form $L_{2k}^s (D_{2k}^s)^{-1} U_{2k}^s$ and $L_{2k+1}^s (D_{2k+1}^s)^{-1} U_{2k+1}^s$, $s = 1, \ldots, l + 1$ that appears during Schur complement computation. From here onwards, we denote an approximation of the Schur complements T_k^s by $\tilde{T}_k^s$ and an approximation of the domain matrix D_k^s by $\tilde{D}_k^s$. Thus, the block **LU** factors for the domain matrix $\tilde{D}_k^{s+1}$ are given as follows

$$\tilde{D}_k^{s+1} = \begin{pmatrix} \tilde{D}_{2k}^s & & U_{2k}^s \\ & \tilde{D}_{2k+1}^s & U_{2k+1}^s \\ \hline L_{2k}^s & L_{2k+1}^s & S_k^{s+1} \end{pmatrix} = \begin{pmatrix} \tilde{D}_{2k}^s & & \\ & \tilde{D}_{2k+1}^s & \\ \hline L_{2k}^s & L_{2k+1}^s & T_k^{s+1} \end{pmatrix} \begin{pmatrix} I & & (\tilde{D}_{2k}^s)^{-1} U_{2k}^s \\ & I & (\tilde{D}_{2k+1}^s)^{-1} U_{2k+1}^s \\ \hline & & I \end{pmatrix}.$$

$$(12)$$

As already mentioned in the section 1, we shall consider three possible approximations as follows:

– **NSSOR**: Here we approximate the Schur complements by setting

$$T_k^j = S_k^j, \quad j = 1, \ldots, l + 1, \quad k = 0, \ldots, 2^j - 1.$$

This can be implemented by calling Algorithm 1 with parameters $lev = l + 1$ and $k = 0$. Since for NSSOR the Schur complements are approximated by the diagonal blocks, we only need to factor these diagonal block to be used

during the solve phase. In Algorithm 1, first the top level separator block is factored, see step (6), then, the next level separator blocks are factored by recursive calls in steps (7) and step (8). Finally the domain matrices at the lowermost level 0 are kept in factored form in step (4).

Algorithm 1. BuildNSSOR(lev, k)

1: INPUT: **A**, $lev = l + 1$
2: OUTPUT: $\tilde{T}$
3: **if** lev=0 **then**
4:    Factor(D_k^0)
5: **else**
6:    Factor(S_k^{lev})
7:    **cilk_spawn** BuildNSSOR($lev - 1$, $2k$)
8:    BuildNSSOR($lev - 1$, $2k + 1$)
9: **end if**

- **NMILUR**: In this method, the Schur complements are approximated such that the preconditioner satisfies the so-called rowsum property (or right filtering on vector of all ones).

$$T_k^1 = S_k^1 - \text{diag}(L_{2k}^0(D_{2k}^0)^{-1}U_{2k}^0 \mathbf{1}) - \text{diag}(L_{2k+1}^0(D_{2k+1}^0)^{-1}U_{2k+1}^0 \mathbf{1}),$$
$$k = 0, \ldots, 2^l - 1$$
$$T_k^{l+1} = S_k^{l+1} - \text{diag}(L_{2k}^l(\tilde{D}_{2k}^l)^{-1}U_{2k}^l \mathbf{1}) - \text{diag}(L_{2k+1}^l(\tilde{D}_{2k+1}^l)^{-1}U_{2k+1}^l \mathbf{1}),$$

where $l > 0, k = 0, \ldots, 2^l - 1$. See Algorithm 2 for implementation details. The steps involved in building the nested preconditioners are outlined in Fig. 2. First of all, the partitioned domain matrices (D_k^0) are factored and are used to construct the Schur complements T_k^1. Afterwards, we have an approximate block **LU** factorization for the domain matrices $\tilde{D}_k^1$ given by (12). Then these approximate factors are used to build the Schur complements T_k^2 which then leads to an approximate block **LU** factorization for the domain matrices $\tilde{D}_k^2$ and so on. These are precisely the steps described in algorithm 2.

- **NFF**: Here the Schur complements are approximated such that the preconditioner satisfies the filtering property on a given filter vector $\mathbf{t}^T = (t_{2k}^l, t_{2k+1}^l)$ as follows

$$\tilde{T}_k^{l+1} = S_k^{l+1} - L_{2k}^l \beta_{2k}^l U_{2k}^l - L_{2k+1}^l \beta_{2k+1}^l U_{2k+1}^l$$
$$\beta_{2k}^l = \text{diag}(\tilde{D}_{2k}^l)^{-1}(U_{2k}^l t_{2k}^l)./(U_{2k}^l t_{2k}^l) \qquad (13)$$
$$\beta_{2k+1}^l = \text{diag}(\tilde{D}_{2k+1}^l)^{-1}(U_{2k+1}^l t_{2k+1}^l)./(U_{2k+1}^l t_{2k+1}^l) \qquad (14)$$

See Algorithm 3 for implementation details.

Once the Schur complements are found and stored in $\tilde{T}$, the solution procedure for solving with these nested preconditioners is identical. This involves calling the subsequent forward and backward sweep routines shown in Algorithms 4
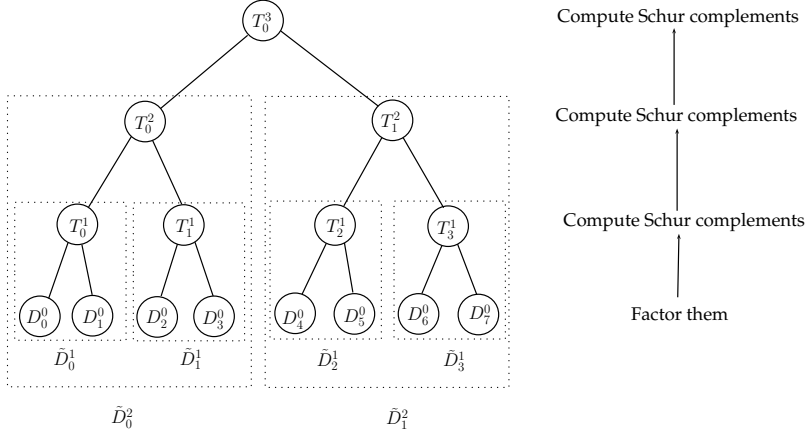
Fig. 2. Steps in building the nested preconditioners

and 5 respectively. Notice here that the solve with the domain matrices $\tilde{D}_{2k}^l$ and $\tilde{D}_{2k+1}^l$ uses the factored form as shown in equation (12). We illustrate the forward sweep procedure. A typical forward sweep is given as follows

$$\begin{pmatrix} \tilde{D}_{2k}^l & & \\ & \tilde{D}_{2k+1}^l & \\ L_{2k}^l & L_{2k+1}^l & \tilde{T}_k^{l+1} \end{pmatrix} \begin{pmatrix} y_{2k}^l \\ y_{2k+1}^l \\ \overline{y}_{2k}^{l+1} \end{pmatrix} = \begin{pmatrix} b_{2k}^l \\ b_{2k+1}^l \\ \overline{b}_k^{l+1} \end{pmatrix}. \tag{15}$$

Notice that the bar notation is used (for example $\overline{b}_k^{l+1}$) to denote the part of the vector corresponding to the separator block. The forward sweep now corresponds to the following steps:

$$\tilde{D}_{2k}^l y_{2k}^l = b_{2k}^l,$$
$$\tilde{D}_{2k+1}^l y_{2k+1}^l = b_{2k+1}^l,$$
$$\tilde{T}_k^{l+1} \overline{y}_{2k}^{l+1} = \overline{b}_k^{l+1} - L_{2k}^l y_{2k}^l - L_{2k+1}^l y_{2k+1}^l.$$

The three steps of the forward sweep procedure correspond to the steps (6), (7), and (8) in Algorithm 4. The backward sweep procedure is similar and is shown in Algorithm 5.

The domain matrices $\tilde{D}_{2k}^l$ and $\tilde{D}_{2k+1}^l$ themselves allow ND representation for $l > 1$, thus, the steps (6) and (7) are recursive calls to the forward sweep and backward sweep algorithms.

Next we consider the sparse matrix vector (SpMV) operation implemented using cilk plus. A cilk plus version of the sparse SpMV is described in Algorithm 6. A typical matrix-vector product is given by

$$\begin{pmatrix} y_{2k}^l \\ y_{2k+1}^l \\ \overline{y}_{2k}^{l+1} \end{pmatrix} = \begin{pmatrix} D_{2k}^l & & U_{2k}^l \\ & D_{2k+1}^l & U_{2k+1}^l \\ L_{2k}^l & L_{2k+1}^l & S_k^{l+1} \end{pmatrix} \begin{pmatrix} x_{2k}^l \\ x_{2k+1}^l \\ \overline{x}_{2k}^{l+1} \end{pmatrix} \tag{16}$$

Algorithm 2. BuildNMILUR(*lev, k*)

1: INPUT: **A**, $lev = l + 1$
2: OUTPUT: $\tilde{T}$
3: **if** $lev = 0$ **then**
4:    Factor (D_k^0)
5: **else**
6:    **cilk_spawn** BuildNMILUR($lev - 1$, $2k$)
7:    BuildNMILUR($lev - 1$, $2k + 1$)
8:    **cilk_sync**
9:    Compute Schur

$$\tilde{T}_k^{lev} = S_k^{lev} - L_{2k}^{lev-1}(\tilde{D}_{2k}^{lev-1})^{-1}(U_{2k}^{lev-1}\mathbf{1}) - L_{2k+1}^{lev-1}(\tilde{D}_{2k+1}^{lev-1})^{-1}(U_{2k+1}^{lev-1}\mathbf{1})$$

   where solve with $\tilde{D}_{2k}^{lev-1}$ and $\tilde{D}_{2k+1}^{lev-1}$ uses the factored form in (12)
10:    Factor($\tilde{T}_k^{lev}$)
11: **end if**

Algorithm 3. BuildNFF(*lev, k*)

1: INPUT: **A**, $lev = l + 1$
2: OUTPUT: $\tilde{T}$
3: **if** $lev = 0$ **then**
4:    Factor(D_k^{lev})
5: **else**
6:    **cilk_spawn** BuildNFF($lev - 1$, $2k$)
7:    BuildNFF($lev - 1$, $2k + 1$)
8:    **cilk_sync**
9:    Compute Schur complements $\tilde{T}_k^{lev}$ for NFF as follows

$$\tilde{T}_k^{lev} = S_k^{lev} - L_{2k}^{lev-1}(\beta_{2k}^{lev-1})L_{2k+1}^{lev-1} - L_{2k+1}^{lev-1}(\beta_{2k+1}^{lev-1})L_{2k+1}^{lev-1}$$

   where β_{2k}^{lev-1} and β_{2k+1}^{lev-1} are given by (13) and (14) respectively.
10:    Set Factor($\tilde{T}_k^{lev}$)
11: **end if**

The above computations correspond to steps (7), (8), and (9) in Algorithm 6. The computations of $D_{2k}^{l-1}x_{2k}^{l-1}$ and $D_{2k+1}^{l-1}x_{2k+1}^{l-1}$ are recursive calls to the SpMV routine. The algorithm presented above leads to spatial and temporal locality of data access while read phase of input vector and write phase of the output vector as illustrated in Figure 3. In this figure, we show that due to recursion a segment of the input and output vector fits in the L3 cache. In the subsequent phase, the data located in the L3 cache is utilized by L2 cache. Thus, we notice the data being used is nearby (spatial locality) and they are accessed frequently (temporal locality).

Algorithm 4. ForwardSweep(lev, k)

1: INPUT: $\tilde{T}, \mathbf{b}, lev = l + 1$
2: OUTPUT: $\mathbf{y}$
3: **if** $lev = 0$ **then**
4:     Solve for y_k^0 in $D_k^0 y_k^0 = b_k^0$
5: **else**
6:     **cilk_spawn** Solve for y_{2k}^{lev-1} in $\tilde{D}_{2k}^{lev-1} y_{2k}^{lev-1} = b_{2k}^{lev-1}$
7:     Solve for y_{2k+1}^{lev-1} in $\tilde{D}_{2k+1}^{lev-1} y_{2k+1}^{lev-1} = b_{2k+1}^{lev-1}$
        cilk_sync
8:     Solve for $\overline{y}_k^{lev}$ in

$$\tilde{T}_k^{lev} \overline{y}_k^{lev} = \overline{b}_k^{lev} - L_{2k}^{lev-1} y_{2k}^{lev-1} - L_{2k+1}^{lev-1} y_{2k+1}^{lev-1}$$

9: **end if**

Algorithm 5. BackwardSweep(lev, k)

1: INPUT: $\tilde{T}, \mathbf{y}, lev = l + 1$
2: OUTPUT: $\mathbf{x}$
3: **if** $lev = 0$ **then**
4:     Set $\overline{x}_k^0 = \overline{y}_k^0$
5:     **cilk_spawn** Find $x_{2k+1}^0 = y_{2k+1}^0 - (D_{2k+1}^0)^{-1} U_{2k+1}^0 \overline{x}_k^0$
6:     Find $x_{2k}^0 = y_{2k}^0 - (D_{2k}^0)^{-1} U_{2k}^0 \overline{x}_k^0$
7: **else**
8:     Set $\overline{x}_k^{lev} = \overline{y}_k^{lev}$
9:     **cilk_spawn** Find $x_{2k+1}^{lev} = y_{2k+1}^{lev} - (\tilde{D}_{2k+1}^{lev})^{-1} U_{2k+1}^{lev} \overline{x}_k^{lev}$
10:     Find $x_{2k}^{lev} = y_{2k}^{lev} - (\tilde{D}_{2k}^{lev})^{-1} U_{2k}^{lev} \overline{x}_k^{lev}$
11: **end if**

---

3   Numerical Experiments

The numerical experiments were performed in double precision arithmetic on a quad core Intel processor i7-2820QM (SandyBridge) with 2.30GHz, 16 GB RAM and three levels of cache: L1 (32kB), L2 (256kB), and L3 (8MB). We used the Intel compiler version 12.1.4 which includes `cilk` keywords as extension to the native compiler for building the NSSOR, NMILUR, and NFF methods.

To solve the Poisson problem and the convection-diffusion problems, we used restarted GMRES with a inner subspace dimension of 500 which is kept rather high to ignore the side effects of restarts. The algorithm is stopped whenever the relative norm

$$\|\mathbf{b} - \mathbf{A}\mathbf{x}_k\|/\|\mathbf{b}\| < 10^{-7}.$$

The exact solution $\mathbf{x}^*$ is generated randomly and the right hand side $\mathbf{b}$ is set to $\mathbf{A}\mathbf{x}^*$. For partitioning and reordering, we use the 2-way ND reordering of METIS [5]. The local subdomains are factored and solved using the multifrontal sparse direct solver UMFPACK [4]. The matrix-vector products with submatrices are computed using the Sparse BLAS library [1]. The compressed row sparse (CRS)
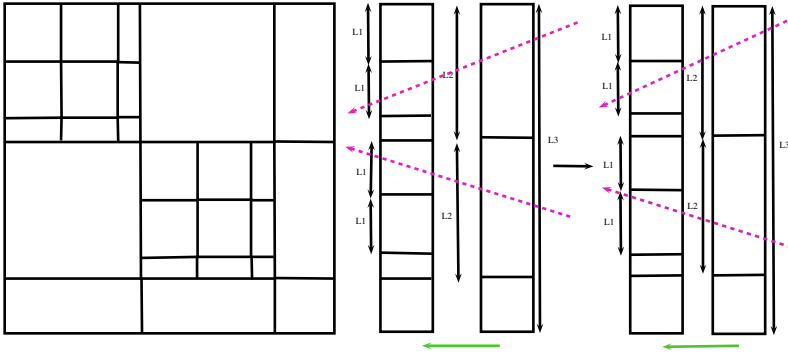
Algorithm 6. SpMV(lev, k)

1: INPUT: $\mathbf{A}$, $\mathbf{x}$, $lev = l + 1$
2: OUTPUT: $\mathbf{y} = \mathbf{Ax}$
3: **if** $lev = 1$ **then**
4:    **cilk_spawn** $y_{2k}^{lev} = D_{2k}^{lev-1} x_{2k}^{lev-1} + U_{2k}^{lev-1} \overline{x}_k^{lev}$
5:    $y_{2k+1}^{lev} = D_{2k+1}^{lev-1} x_{2k+1}^{lev-1} + U_{2k+1}^{lev-1} \overline{x}_k^{lev}$
6: **else**
7:    $\overline{y}_k^{lev} = L_{2k}^{lev-1} x_{2k}^{lev-1} + L_{2k+1}^{lev-1} x_{2k+1}^{lev-1} + S_k^{lev} \overline{x}_k^{lev}$
8:    **cilk_spawn** $y_{2k}^{lev-1} = D_{2k}^{lev-1} x_{2k}^{lev-1} + U_{2k}^{lev-1} \overline{x}_k^{lev}$
9:    **cilk_spawn** $y_{2k+1}^{lev-1} = D_{2k+1}^{lev-1} x_{2k+1}^{lev-1} + U_{2k+1}^{lev-1} \overline{x}_k^{lev}$
10: **end if**

Fig. 3. Illustration of cache utilization during the read phase of the input vector and write phase of the output vector. The dotted lines indicate the data flow in cache hierarchy.

storage scheme is an efficient storage scheme for a matrix without sufficient block structure. For matrices reordered after nested dissection reordering, we have several blocks per block row. This requires maintaining separate row pointers for each of the blocks within the same row. The count of total row indices increases like $O(n\sqrt{P})$, P being the number of threads/partitions. On the other hand, for the coordinate storage format, the number of indices used to store the non-zero entries remain independent of the number of blocks. Hence, instead of the CRS, the coordinate storage format is used to store the $\mathbf{L}$, $\mathbf{D}$, $\mathbf{U}$, and $\mathbf{T}$ matrices. These matrices are stored as an array of structs that encode the binary tree representation; the indices $2i$ and $2i + 1$ being the left and right subdomains of the domain indexed i. The scalar products are computed using the recently introduced elemental functions of `cilk plus` [3].

3.1   Convection Diffusion Problem

We consider the following boundary value problem

$$\begin{aligned}
\mathrm{div}(\mathbf{a}(x)u) - \mathrm{div}(\kappa(x)\nabla u) &= f \ \text{ in } \ \Omega, \\
u &= 0 \ \text{ on } \ \partial\Omega_D, \\
\tfrac{\partial u}{\partial n} &= 0 \ \text{ on } \ \partial\Omega_N,
\end{aligned} \tag{17}$$

where $\Omega = [0,1]^n$ ($n = 2$, or 3), $\partial\Omega_N = \partial\Omega \setminus \partial\Omega_D$. The velocity vector $\mathbf{a}$ and the tensor κ are the given coefficients of the partial differential operator. The domain is unit square in 2D and unit cube in 3D. The equation is discretized using the cell-centered finite volume scheme.

We consider following three test cases:

Poisson: Here $\kappa = 1$ and $\mathbf{a} = 0$. In Table 2 we observe that NSSOR is the most efficient preconditioner since it leads to the lowest execution time in comparison to both NMILUR and NFF. The sharp decrease in the setup time with increasing number of cores for all the methods is both due to parallelism and smaller subdomain sizes incurring lesser work with increasing number of subdomains. For one and two cores, we keep the number of subdomains equal to two, thus the iteration count remains the same, but with four subdomains using four threads, we see a significant increase in the iteration count for both NSSOR and NMILU; the iteration count for NFF increases slightly. We observe a speedup of roughly 1.5 for both NSSOR and NFF. The iteration count for NMILUR increases so much that the solve time increases for four subdomains compared to that for two subdomains.

Table 2. Number of iterations and execution times for Poisson problem on $60 \times 60 \times 60$ grid

cores	NSSOR				NMILUR				NFF			
	its	setup	solve	total	its	setup	solve	total	its	setup	solve	total
1	29	114s	15s	129s	75	116s	39s	155s	13	137s	6s	143s
2	29	65s	11s	76s	75	65s	31s	96s	13	77s	5s	82s
4	34	10s	10s	20s	135	11s	44s	55s	15	28s	4s	32s

Skyscraper Problems: For this case, the velocity $\mathbf{a} = 0$ and the tensor $\kappa(x)$ is isotropic and discontinuous as follows. The domain contains many zones of high permeability which are isolated from each other. Let $\lfloor x \rfloor$ denote the greatest integer less than x. In 3D, we have

$$\kappa(x) = \begin{cases} 10^3 * (\lfloor 10 * x_2 \rfloor + 1), & \text{if } \lfloor 10 * x_i \rfloor = 0 \bmod(2), \ i = 1, 2, 3, \\ 1, & \text{otherwise.} \end{cases}$$

In Table 3, the numerical experiments with this test case are shown. For this problem NFF converges faster than NSSOR and NMILUR. The iteration count

Table 3. Number of iterations and execution times for skyscraper problem on $60 \times 60 \times 60$ grid

cores	NSSOR				NMILUR				NFF			
	its	setup	solve	total	its	setup	solve	total	its	setup	solve	total
1	96	115s	50s	165s	75	116s	39s	155s	19	135s	10s	145s
2	106	63s	43s	76s	75	65s	31s	96s	19	77s	7s	84s
4	> 1000	9s	> 344s	20s	135	11s	44s	55s	44	28s	13s	41s

for NFF is lowest, however, we do see a sharp increase in the iteration count for NFF when the number of subdomains increase from 2 to 4.

Convective Skyscraper Problems: These problems are similar to the skyscraper problems, but now the velocity field is changed to $\mathbf{a} = (1000, 1000, 1000)^T$.

The numerical experiments for this test case are shown in Table (4). Once again NFF has the lowest iteration count followed by NSSOR and NMILUR. For this case, we do not see a sharp jump in the iteration count when the number of subdomains increases from 2 to 4. The setup phase of NFF is large due to the exact factorization of the subdomains. But as the number of subdomains increase, we see a sharp decrease in the setup cost. The setup times of the new methods are high because the subdomain size is large for 2-4 subdomains. The setup times are expected to decrease drastically as the numbers of subdomains increases.

Table 4. Number of iterations and execution times for convective skyscraper problem on $60 \times 60 \times 60$ grid

cores	NSSOR				NMILUR				NFF			
	its	setup	solve	total	its	setup	solve	total	its	setup	solve	total
1	81	113s	42s	155s	230	117s	131s	248s	18	134s	9s	143s
2	82	63s	33s	76s	241	65s	108s	173s	18	76s	7s	83s
4	152	9s	49s	58s	461	11s	189s	199s	21	27s	6s	33s

4   Conclusion

A class of non-overlapping domain decomposition preconditioners based on 2-way nested dissection reordering is implemented. For multithreading, we used `cilk plus` with new elemental function features. We have presented preliminary scalability results indicating that the nested filtering factorization preconditioner is a promising method. In particular, the NSSOR and NFF preconditioners can be applied to general symmetric positive definite problems. Note that for Poisson and convection-diffusion problems, geometric or algebraic multigrid could be used. However, for more general problems, the NFF preconditioner may be preferred especially if the following improvements are implemented:

– introduction of parallelism in the levels higher up the tree-for example, it is possible to spawn more threads during Schur complement computations and solve phases;
– better cache utilization by implementing a reordering based on a space filling curve for more efficient sparse matrix-vector computations.

References

1. NIST Sparse BLAS, `http://math.nist.gov/spblas/`
2. HYPRE, `http://acts.nersc.gov/hypre/`
3. Cilk plus, online documentation at `http://software.intel.com/file/40297`
4. Davis, T.A.: Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. ACM Transactions on Mathematical Software 30(2), 196–199 (2004)
5. METIS graph partitioner, `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`
6. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. In: International Conference on Parallel Processing, pp. 113–122 (1995)
7. Karypis, G., Kumar, V.: Analysis of Multilevel Graph Partitioning. In: Supercomputing (1995)
8. Karypis, G., Kumar, V.: Multilevel k-way Partitioning Scheme for Irregular Graphs. J. Parallel Distrib. Comput. 48(1), 96–129 (1998)
9. Grigori, L., Kumar, P., Nataf, F., Wang, K.: A class of multilevel parallel preconditioning strategies. HAL Report RR-7410 (2010) (2010), `http://hal.inria.fr/inria-00524110_v1/`
10. George, J.A.: Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis 10(2), 345–363 (1973)
11. Kumar, P.: A class of preconditioning techniques suitable for partial differential equations of structured and unstructured mesh. PhD thesis (2010), `http://www.sudoc.fr/147701619`
12. Weiler, W., Wittum, G.: Parallel Frequency Filtering. Computing 58, 303–316 (1997)
13. Wagner, C.: Tangential frequency filtering decompositions for symmetric matrices. Numer. Math. 78(1), 119–142 (1997)
14. Wagner, C.: Tangential frequency filtering decompositions for unsymmetric matrices. Numer. Math. 78(1), 143–163 (1997)
15. Wittum, G.: Filternde Zerlegungen-Schnelle Loeser fur grosse Gleichungssysteme. Teubner Skripten zur Numerik Band 1. Teubner-Verlag, Stuttgart (1992)

16. Axelson, O., Polman, B.: A robust preconditioner based on algebraic substructuring and two level grids. In: Hackbusch, W. (ed.) Robust multi-grid methods, NNFM, Bd.23. Vieweg-Verlag, Braunschweig
17. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS publishing company, Boston (1996)
18. Kettler, R.: Analysis and comparison of relaxation schemes in robust multigrid and preconditioned conjugate gradient methods. In: Multigrid Methods. Lecture notes in mathematics, vol. 960, pp. 502–534 (1982)
19. Notay, Y.: AGMG: Agregation based AMG,
    `http://homepages.ulb.ac.be/~ynotay`