J Supercomput (2016) 72:2259–2282
DOI 10.1007/s11227-016-1728-5

Parallel multilevel recursive approximate inverse techniques for solving general sparse linear systems

Antonios T. Makaratzis[1] · **Christos K. Filelis-Papadopoulos**[1] · **George A. Gravvanis**[1]

Published online: 5 May 2016

Abstract In this article, a new parallel multilevel algebraic recursive generic approximate inverse solver (PMARGAIS) is proposed. PMARGAIS utilizes the parallel modified generic factored approximate sparse inverse (PMGenFAspI) matrix technique designed for shared memory parallel systems. PMARGAIS requires a block independent set reordering scheme, to create a hierarchy of levels. A modified block breadth first search (MBBFS) is proposed for reducing memory requirements and retaining load balancing. The SVD method is used to compute the inverse of the independent blocks that are formed from the reordering scheme, and computes accurately the Schur complement that is used as a coefficient matrix on the next level, resulting in a hybrid direct-iterative method for large linear systems. The solution of the linear system at the last level is performed with the parallel explicit preconditioned BiCGSTAB method in conjunction with the PMGenFAspI matrix. The parallelization of the proposed methods uses the vector units of modern CPUs. Implementation details are provided and numerical results are given demonstrating the applicability and effectiveness of the proposed schemes.

Keywords Parallel modified factored approximate sparse inverses · Parallel multilevel algebraic recursive generic approximate inverse solver · Parallel explicit preconditioned bi-conjugate gradient stabilized method · Shared memory parallel systems · Vector units

✉ George A. Gravvanis
ggravvan@ee.duth.gr

[1] Department of Electrical and Computer Engineering, School of Engineering, Democritus University of Thrace, University Campus, Kimmeria, 67100 Xanthi, Greece

1 Introduction

Let us consider a sparse linear system

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

where A is the coefficient matrix, b is the right-hand side vector and x is the solution vector of the linear system. Preconditioned Krylov subspace iterative methods are amongst the most widely used iterative methods. The effectiveness of these iterative methods relies on the use of effective preconditioning schemes that reduce the number of required iterations and in many cases ensure converge, cf. [3,15,16,18,22–24]. Approximate inverses have been extensively used as preconditioners with iterative methods, cf. [3,7,8,10,13,15,16,18]. The approximate inverses possess inherent parallelism, cf. [10,13,15,16], and thus can be effectively used on parallel systems. Recently, a generic class of approximate inverses has been proposed that can handle any sparsity pattern of the coefficient matrix, cf. [14]. By redesigning the Generic Approximate Banded Inverse algorithm, cf. [14], and utilizing approximate inverse sparsity patterns], derived from patterns of sparsified matrices (PSMs), cf. [7,8], the generic approximate sparse inverse (GenAspI) algorithm as well as the generic factored approximate sparse inverse (GenFAspI) algorithm were proposed, cf. [10].

Multilevel techniques have been proposed in the recent years by many researchers, cf. [4,5,23,24]. These methods utilize reordering schemes and techniques from domain decomposition methods, cf. [4,5,19,21,23,24,27]. Nested Grids ILU decomposition (NGILU), cf. [4], multilevel recursive incomplete LU factorization (MRILU), cf. [5], algebraic recursive multilevel solver (ARMS), cf. [23], and multilevel algebraic recursive generic approximate inverse solver (MARGAIS), cf. [10] are multilevel techniques that have been proposed in recent years.

Herewith, parallel schemes are proposed for shared memory parallel systems, using the OpenMP environment, cf. [6]. The parallel modified generic factored approximate sparse inverse (PMGenFAspI) method is proposed, which is a parallel version of the MGenFAspI method, cf. [10], using vector units, cf. [17]. The proposed parallel multilevel solver, namely parallel multilevel algebraic recursive generic approximate inverse solver (PMARGAIS), utilizes a modified reordering scheme that is based on block breadth first search (BBFS), cf. [23]. The coefficient matrix of the system is reordered such that the upper left block is a block diagonal matrix. The inversion of the block diagonal matrix is performed in parallel, utilizing the SVD method, cf. [11,12], for each block. The modified reordering scheme (MBBFS) ensures that the dimensions of each block remains small, resulting to less memory requirements and balanced computational work during the inversion of the block diagonal matrix. The Schur complement is formed explicitly to be used as the coefficient matrix on the next level. This process is repeated until the linear system of the last level is small enough to be solved efficiently. The parallel explicit preconditioned bi-conjugate gradient stabilized (PEPBiCGSTAB) method, cf. [28], is parallelized for shared memory parallel systems in conjunction with AVX vector units, cf. [17], and used to solve the reduced order linear

system of the last level. The explicit formation of the Schur complement and the exact inversion of the block diagonal inverse, leads to a hybrid direct-iterative method.

The AVX units are vector units used for carrying out concurrent computations to multiple data following the SIMD model, cf. [17]. Efforts have been concentrated by other researchers to facilitate efficient processing of problems that involve matrix and vector computations at the hardware level, cf. [1,25,26]. These efforts involve the design of specialized units, based on reversible logic synthesis, to carry out efficiently such types of concurrent computations, cf. [1,25,26].

In Sect. 2, the PMGenFAspI method is presented along with implementation details. In Sect. 3, the PMARGAIS method based on the modified reordering scheme (MBBFS) is presented, along with discussions on the performance improvements. Furthermore, implementation details for the parallel inversion of the block diagonal matrix and the computation of the Schur complement are given. Further, the parallel EPBiCGSTAB method based on AVX units is given. In Sect. 4, numerical results presenting the performance and applicability of the proposed schemes are given.

2 Parallel modified generic factored approximate sparse inverse (PMGenFAspI)

The MGenFAspI matrix, cf. [10], can be computed using the following decomposition, cf. [2,12,20,22]:

$$A = LU \Leftrightarrow A^{-1} = U^{-1}L^{-1} \Leftrightarrow M = GH \tag{2}$$

where $M = A^{-1}$, $G = U^{-1}$ and $H = L^{-1}$. The factors L and U are obtained from Incomplete LU decomposition, cf. [2,20,22]. The sparsity patterns of the factors G and H are computed by sparsifying the triangular factors L and U using a predetermined drop tolerance (drptol). The resulting sparsified matrix is then raised to a predefined power or level of fill (lfill). The sparsity pattern is based on powers of sparsified matrices (PSM's), cf. [7,8]. Hence, to compute the elements of the G and H factors we have to solve the following systems, cf. [10]:

$$\left\{ \begin{matrix} UG^{\text{lfill}}_{\text{drptol}} = I \\ LH^{\text{lfill}}_{\text{drptol}} = I \end{matrix} \right\} \Leftrightarrow \left\{ \begin{matrix} Ug_{:,j} = e_{:,j} \\ Lh_{:,j} = e_{:,j} \end{matrix} \right\}, \quad 1 \leq j \leq n \tag{3}$$

where lfill is the level of fill used to compute the sparsity pattern of the approximate inverse and drptol is the threshold for retaining elements in the initial sparsity pattern of the approximate inverse, cf. [7,8,10], while $g_{:,j}$ and $h_{:,j}$ are the elements of the jth column of the triangular factors of the approximate inverse and $e_{:,j}$ are the elements of the jth column of the identity matrix. During the computation, the elements $g_{:,j}$ and $h_{:,j}$ are stored in a dense vector iw to prevent column search for elements in the sparse format matrices G and H. The elements of the jth column of the identity matrix $e_{:,j}$ are stored in a dense vector e. Each nonzero element of

the H and G factors of the MGenFAspI method can be computed by the following equations:

$$H(k,j) = \frac{I(k,j) - L(k,1:k-1) * H(1:k-1,j)^T}{L(k,k)},$$
$$k = \mathrm{j}, \ldots, n, (k,j) \in H^{\mathrm{lfill}}_{\mathrm{drptol}} \tag{4}$$

$$G(k,j) = \frac{I(k,j) - U(k,k+1:n)^T * G(k+1:n,j)}{U(k,k)},$$
$$k = j, \ldots, 1, (k,j) \in G^{\mathrm{lfill}}_{\mathrm{drptol}}. \tag{5}$$

where I is the identity matrix.

The parallel computation of the MGenFAspI process can be performed efficiently due to the fact that the computation of each column of the factors G and H is not related to the computation of other columns. Each processor is responsible for computing a group of columns of the approximate inverse without any communications. The PMGenFAspI method has been further modified to utilize AVX units accelerating the computation of the involved inner products. Initially the values of a register are set to zero. Then, the values of the involved vectors residing in the memory are transferred in groups of four values to two registers. The computation of the products is performed concurrently and the respective results are accumulated to the register retaining the partial sums. The procedure is repeated until all the elements have been accumulated, resulting in four partial sums. The inner product is computed by adding the four partial sums. In case the number of nonzero elements is not a multiple of four, the remaining elements are accumulated independently. The parallel modified generic factored approximate sparse inverse scheme in conjunction with AVX units is described by the following algorithmic scheme:

PMGenFAspI Algorithm with AVX

Let G and H be the approximate inverse factors with drptol drop tolerance and lfill levels of fill

Parallel For i=0,…,n-1
For j∈H(:,i) with j≥i
sum=0
For k∈L(j,:) with k<j **with step** 4
Load 4 double numbers from L(j,:) **in main memory to register** *xr1*
Load 4 double numbers from iw **in main memory to register** *xr2*
xr3=fmadd(xr3,xr1,xr2)
End For (k)
sum=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)
Add remaining products L(j,k)*iw(k) **to** sum
iw(j)=(e(j,i)-sum)/L(j,j)
End For (j)
H(:,i)=iw – Sparse set (A column of the H factor)
iw(j∈H(:,i) with j≥i)=0 – Sparse set to zero
For j∈G(:,i) with j≤i in reverse order
sum=0
For k∈U(j,:) with k>j **with step** 4
Load 4 double numbers from U(j,:) **in main memory to register** *xr1*
Load 4 double numbers from iw **in main memory to register** *xr2*
xr3=fmadd(xr3,xr1,xr2)
End For (k)
sum=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)
Add remaining products U(j,k)*iw(k) **to** sum
iw(j)=(e(j,i)-sum)/U(j,j)
End For (j)
G(:,i)=iw – Sparse set (A column of the G factor)
iw(j∈G(:,i) with j≤i)=0 – Sparse set to zero
End For (i)

where $fmadd(xr3, xr1, xr2)$ is the fused multiply add operation $xr3 = xr3 + xr1 * xr2$, where $xr1$, $xr2$ and $xr3$ are vectors consisting of four double-precision floating point numbers. The iw vector is the work vector used to store temporarily the elements of the ith column of each of the factors of the approximate inverse, while the vector e is the ith column of the identity matrix.

It should be noted that the length of the vectors iw and e is multiplied with the number of the processors that are being used, so that each processor uses a different part of the vectors. The elements of the vectors are determined as follows:

$$\mathbf{iw}\,(\mathbf{cur\_tid} * \mathbf{n} + \mathbf{k}) : (iw\,(k) \text{ of cur\_tid processor}) \tag{6}$$

$$\mathbf{e}\,(\mathbf{cur\_tid} * \mathbf{n} + \mathbf{k}) : (\text{e (k) of cur\_tid processor}) . \tag{7}$$

where n is the number of the rows of the factors G and H, cur_tid=0,. . ., nprocs-1 is the id number of each processor and nprocs is the total number of processors.

3 Parallel multilevel algebraic recursive generic approximate inverse solver (PMARGAIS)

3.1 Modified block breadth first search (MBBFS)

Independent Sets are a crucial component of multilevel methods. An Independent Set is composed of unknowns that are decoupled between them and can be handed independently or simultaneously without affecting other unknowns of the linear system, cf. [23,24]. Recently, a reordering scheme was introduced for the algebraic multilevel recursive solver (ARMS), cf. [23], based on Breadth First Search algorithm, utilizing a threshold parameter which restricts unknowns that have lesser relative diagonal dominance in their respective row, to join the independent set, cf. [23]. The block independent sets is a generalization of the Independent sets, where groups of unknowns are decoupled. The unknowns belonging to the Block Independent sets are numbered first and the interface unknowns are numbered last. Hence, the upper left block of the permuted coefficient matrix has a block diagonal structure.

The modified block breadth first search (MBBFS) algorithm reduces the memory requirements of the block diagonal inverse by retaining the dimension of the (dense) blocks to a predefined small number. For the BBFS algorithm, grouping of nodes into independent sets stops after a whole level of neighbors has been added and the block size has exceeded the predetermined block size. This technique results in blocks of much bigger size than the predetermined one, especially in the case of matrices with large number of nonzero elements per row. In contrast the MBBFS scheme stops the insertion of unvisited nodes to a block independent set when the predetermined block size is reached. This technique results in blocks of size equal to the predetermined one or smaller when there are not enough neighboring nodes, thus giving an upper limit to the memory requirements for computing the inverse of the block diagonal matrix. The MBBFS scheme improves the balance of computational work between the CPUs, since the dimensions of each block are almost the same.

It should be stated that the algorithm also returns a vector s storing the starting and ending point of each block, to handle the blocks independently:

$$s(\text{block}) = \sum_{j=1}^{\text{block}} \text{block\_size}(j), \ \text{block} = 1, \ldots, \text{number\_of\_blocks} \tag{8}$$

where block_size(j) is the size of each block and $s(0) = 0$. Utilizing the vector s, the following information can be obtained:

- **s(i-1):** the row and the column of the block diagonal matrix where the first row and column of the ith block are located.
- **s(i)-1:** the row and the column of the block diagonal matrix where the last row and column of the ith block are located.
- **s(i)-s(i-1)**: the order of the ith block.

The modified block breadth first search algorithm is described by the following algorithmic scheme:

MBBFS Algorithm

Let R be the reordering vector.

Set all the vertices in U

Compute relative diagonal dominance weights for every row of the matrix A

R={}

While U≠{}

B={u} if u is not marked as excluded

For all neighbors N_i of u up to the bsize level set

If $|B|+|N_i| \le$ bsize

B=B∪N_i and remove from U if they have relative diagonal dominance lesser than the prescribed tolerance, else mark them as excluded and remove from U.

Else

Let $W \subset N_i$ such that $|W|+|B| \le$ bsize.

B=B∪W and remove from U if they have relative diagonal dominance lesser than the prescribed tolerance, else mark them as excluded and remove from U.

End If

End For

Mark all neighbors of block B to be excluded and remove from U

Reverse the order in B

R=R∪B

End While

Append the excluded nodes in the end of R

It should be mentioned that in case the number of vertices grouped in a block is smaller than the prescribed value (bsize) and no more neighboring vertices exist, then the block is retained as is.

3.2 Parallel multilevel approximate inverse solver

Let us consider the reordering matrix P computed with the MBBFS algorithm. The reordered block matrix K is as follows, cf. [10]:

$$K = PAP^T = \begin{bmatrix} B & C \\ D & E \end{bmatrix} = \begin{bmatrix} I & 0 \\ DB^{-1} & I \end{bmatrix} \begin{bmatrix} B & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & B^{-1}C \\ 0 & I \end{bmatrix} \tag{9}$$

where $S = E - DB^{-1}C$ is the Schur complement of the block matrix A and B is a block diagonal matrix. The inverse of K can be computed as follows:

$$K^{-1} = \begin{bmatrix} B & C \\ D & E \end{bmatrix}^{-1} = \begin{bmatrix} I & -B^{-1}C \\ 0 & I \end{bmatrix} \begin{bmatrix} B^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -DB^{-1} & I \end{bmatrix}. \tag{10}$$

The inverse of the block matrix can be computed by inverting the matrices B and S. Matrix B is a block diagonal matrix, thus the inverse B^{-1} is a block diagonal matrix. The order of the blocks is small and the exact inverse matrices can be computed using the SVD method separately on each block. The inverse matrix B^{-1} that is computed with this technique is the exact inverse of matrix B. The inverse of each block can be computed independently, thus each processor is responsible for inverting a different group of blocks. The number of nonzero elements is small compared to the order of matrix B, thus the inversion of the blocks does not require substantial computational work or high memory requirements.

Computing the inverse of the Schur complement is less expensive due to its reduced order compared to the matrix K, and can be computed approximately with the PMGenFAspI method. This process leads to the computation of a two-level approximate inverse. The two-level process could be recursively applied to invert the resulting Schur complement, leading to a multilevel scheme. The scheme is applied until the Schur complement is sufficiently small and can be inverted efficiently, or no more independent sets exist. The last Schur complement can be approximately inverted using the PMGenFAspI method.

In practice, it is inefficient to compute an approximate inverse explicitly using this multilevel technique, because the involved operations tend to have increasing number of nonzero elements, cf. [10]. Instead the linear system (1) can be solved in block form.

The equivalent expression for the reordered system is of the following form:

$$\left(PAP^{\mathrm{T}}\right) Px = Pb \Leftrightarrow Kx\prime = b' \Leftrightarrow x\prime = K^{-1}b' \tag{11}$$

where $K = PAP^{\mathrm{T}}$, $x' = Px$ and $b' = Pb$. Using Eq. (10) we derive the following:

$$\begin{bmatrix} x_i \\ x_r \end{bmatrix} = K^{-1} \begin{bmatrix} b_i \\ b_r \end{bmatrix} = \begin{bmatrix} I & -B^{-1}C \\ 0 & I \end{bmatrix} \begin{bmatrix} B^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -DB^{-1} & I \end{bmatrix} \begin{bmatrix} b_i \\ b_r \end{bmatrix} \tag{12}$$

where the subscript i denotes the solution and the right-hand side corresponding to the nodes associated with the independent sets and the subscript r denotes the solution and

the right-hand side corresponding to the rest of the nodes. The equivalent expression of (12) is as follows:

$$\begin{bmatrix} x_i \\ x_r \end{bmatrix} = \begin{bmatrix} B^{-1}b_i - B^{-1}CS^{-1}(-DB^{-1}b_i + b_r) \\ S^{-1}(-DB^{-1}b_i + b_r) \end{bmatrix} \tag{13}$$

or equivalently

$$x_i = B^{-1}b_i - B^{-1}Cx_r. \tag{14}$$

$$x_r = S^{-1}(-DB^{-1}b_i + b_r) \tag{15}$$

The solution vector x_i can be computed directly since B^{-1} is known explicitly and x_r is computed by (15). Hence, x_i is computed with three matrix vector multiplications and with one vector subtraction. These computations are executed in parallel using AVX units. Nevertheless, the solution vector x_r is computed by solving the linear system:

$$Sx_r = (-DB^{-1}b_i + b_r). \tag{16}$$

This linear system can be solved with the PEPBiCGSTAB in conjunction with the PMGenFAspI method. This process leads to a two-level hybrid direct-iterative scheme for solving linear systems. The two-level process could be recursively applied to the linear system (16), leading to a multilevel scheme for the computation of the solution vector x. The multilevel scheme is depicted in Fig. 1. The solution vector x_i of the last

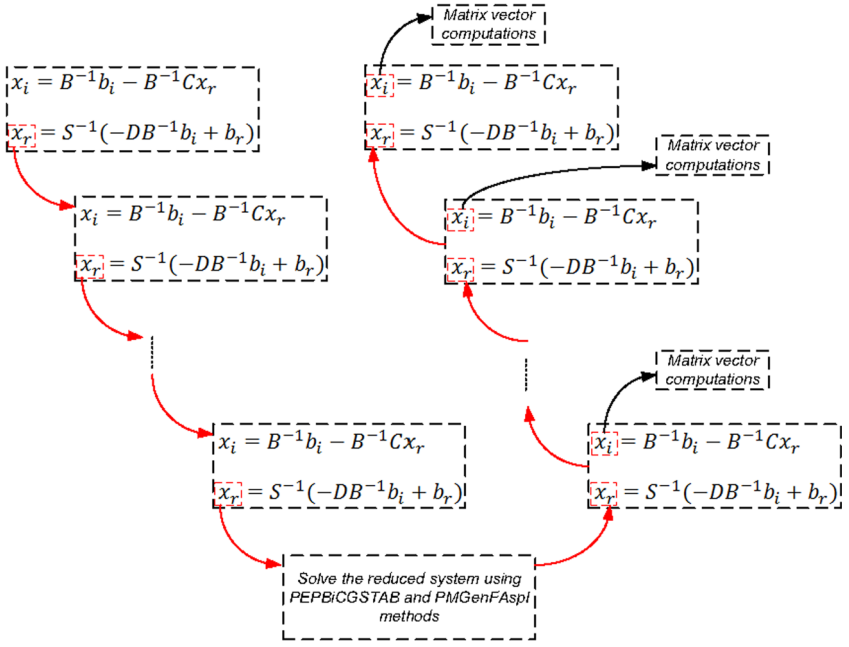

Fig. 1 Multilevel solution of a linear system through recursive solution of continuously smaller Schur complement linear systems along with the linear systems corresponding to the block independent sets

level is computed using the vector x_r. The vector $x = P^T[x_i^T \; x_r^T]^T$ is returned as the solution vector x_r of the previous level.

In the case where the singular values of B are close to machine precision, the method might converge in more than a single iteration, due to the rounding errors. Moreover, singular values close to the machine precision are set explicitly to zero. This is also true in the case where the prescribed tolerance is close to machine precision. In such cases, the multilevel process is used as a preconditioner to the Richardson iterative method,

$$x_{k+1} = x_k + P\text{MARGAIS}(\text{A},r_k) \tag{17}$$

where $r_k = b - Ax_k, k = 0, 1, 2\ldots$ is the residual vector. The scheme is repeated until the solution of the linear system is acquired to the prescribed tolerance. The multilevel scheme is a hybrid direct-iterative method.

The parallel multilevel algebraic recursive generic approximate inverse solver (PMARGAIS) can then be described by the following algorithmic scheme:

Algorithm PMARGAIS

x=PMARGAIS(A,b,level)

If level≠last_level

Setup the reordering matrix P using MBBFS method.

Form $K=PAP^T$ and b'=Pb in parallel.

Parallel computation of the exact inverse B^{-1} using SVD method.

Parallel computation of Schur Complement: $S=E-DB^{-1}C$.

Compute x_r=PMARGAIS(S,$-DB^{-1}b_i+b_r$,level+1)

Compute $x_i=B^{-1}b_i-B^{-1}Cx_r$ in parallel.

Compute $x=P^T[x_i^T \; x_r^T]^T$

Else If level=last_level

Parallel solution of the system Ax=b using PEPBiCGSTAB method in conjunction with the PMGenFAspI technique, based on AVX units.

Return of the solution vector x to the previous level.

End If

The PEPBiCGSTAB method, cf. [13,28], using AVX units, is presented in the "Appendix".

It should be noted that the parallel computations of the method use the cache blocking technique. Thus, each computation is tiled to segments such that eight double-precision floating point numbers are transferred to the cache memory, since the L1 cache line of the systems used is 512 bits (64 bytes).

3.3 Parallel inversion of block diagonal matrix

For the exact inversion of the block diagonal matrix B, each block of the matrix is inverted using the SVD method. The dimensions of each block are kept small, thus

the computational work as well as the memory requirements required for inversion are kept reduced. Using the SVD method, the block that is inverted takes the following form, cf. [11,12]:

$$B_{\text{sub}} = U \Sigma V^T \tag{18}$$

where U and V are orthogonal matrices, thus their inverses are equal to their transposes, and Σ is a diagonal matrix, cf. [11,12]. Hence, the inverse of the block B_{sub} can be computed as follows:

$$(B_{\text{sub}})^{-1} = V \Sigma^{-1} U^T. \tag{19}$$

It should be stated that in case the dimension of the block is equal to one, then the inversion is trivial and does not require the SVD method. After the computation of the inverse of each blocks, they are stored in B^{-1} assembling the complete inverse matrix. The mapping from local to global positions in the inverse is realized through the vector s, which is obtained by the MBBFS method. Then, the exact inverse of B is computed by the following algorithmic compact scheme:

Parallel computation of B^{-1}

Let B^{-1} be the inverse of the block diagonal matrix B.

$tA=PAP^T$

Parallel For block=1,…,num_of_sets

start_p=s(block-1), end_p=s(block)-1, sub_size=end_p-start_p

B_{sub}=tA(start_p:end_p,start_p:end_p)

If sub_size=1

B^{-1}(start_p)=1/B_{sub}(0,0)

Else

v (:,:)=0.0; q(:,:)=0.0; u(:,:)=B_{sub}(:,:)

SVD (q,u,v)

Set the singular values that are close to the machine precision equal to zero.

B^{-1}(start_p:end_p,start_p:end_p)=(v)(q^{-1})(u^T)

End If

End For

The blocks of matrix B have arbitrary form, thus their inverses are generally dense. To form the matrix B^{-1} the memory requirements have to be a priori computed. The vector ss is retaining the memory requirements for each block and its elements are computed as follows:

$$\text{ss(block)} = \sum_{j=1}^{\text{block}} [s(j) - s(j-1)]^2, \quad \text{block} = 1, \ldots, \text{number\_of\_blocks} \tag{20}$$

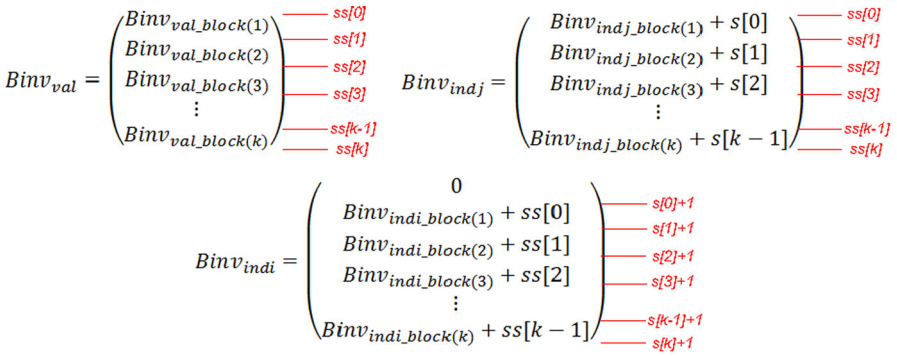

Fig. 2 Compressed sparse row storage format (three vector variant) of B^{-1}

where ss(0) = 0. The values of the vector ss denote the number of the nonzero elements that are stored in the inverse matrix B^{-1}. The block matrix B^{-1} is stored in compressed sparse row (three vector variant) storage format using the values of the vector ss, the values of the inverses of the blocks, as well as the column indexes computed from the s vector, as depicted in Fig. 2.

The inverse of each block of the matrix B can be computed with the multiplication $(v)(q^{-1})(u^T)$ where u, q and v are computed by the SVD decomposition. The triple matrix multiplication involves dense computations that can be parallelized with AVX units. Parallelization with AVX units is fine grained, thus the inner summation loop is parallelized. The dense matrix multiplication with AVX units can be described by the following algorithmic procedure:

Dense Matrix Multiplication Algorithm with AVX

For i=1,...,sub_size

  For k=1,...,sub_size

    Load 4 times the double number A(i,k) **to register** *xr1*

    For j=1,...,sub_size **with step** 4

      Load 4 double numbers from B(k,j...j+3) **in main memory to register** *xr2*

      Load 4 double numbers from C(i,j...j+3) **in main memory to register** *xr3*

      xr3=fmadd(xr3,xr1,*xr2*)

      Store the 4 double numbers from register *xr3* **to** C(i,j...j+3)

    End For (j)

    Compute remaining C(i,j)= C(i,j)+A(i,k)*B(k,j)

  End For (k)

End For (i)

where $fmadd(xr3, xr1, xr2)$ is the fused multiply add operation $xr3 = xr3 + xr1 * xr2$, where $xr1$, $xr2$ and $xr3$ are vectors consisting of four doubles.

The Schur complement is computed by the following equation:

$$S = E - DB^{-1}C. \tag{21}$$

The computation of the Schur complement requires sparse matrix operations, which are computationally intensive. The required sparse matrix multiplications can be parallelized efficiently since each processor computes the assigned number of rows independently. Sparse matrix multiplications are two-step operations: initially the possible number of nonzero elements should be evaluated and then the values of the resulting nonzero elements can be computed. It should be mentioned that the computation of each row is performed using a dense work vector (w) and a list, which stores the nonzero values as well as their positions. In case of multiple processors, the dimension of this vector is computed by multiplying the number of columns of the first sparse matrix with the number of processors. Hence, each processor uses a different part of the vector.

4 Numerical results

In this section, the performance of parallel multilevel algebraic recursive generic approximate inverse solver (PMARGAIS) is examined for solving various problems. The execution time is given in ss.hh (seconds.hundreds) and the overall gain corresponds to the time of the serial execution divided with the time of the serial or parallel execution that includes the use of AVX units. The problems ATMOSMODD, tmt_unsym and cage13 were obtained from the University of Florida Sparse Matrix Collection, cf. [9]. The Poisson problem in two and three dimensions can be described by the following PDE:

$$-\Delta u = f, (x_1, x_2, \ldots, x_d) \in \Omega = [0, 1]^d \tag{22}$$

$$u = 0, (x_1, x_2, \ldots, x_d) \in \partial\Omega \tag{22.a}$$

where $\partial\Omega$ denotes the boundary of Ω and d denotes the number of the space variables. The above PDE is discretized with the five point stencil finite differences method in two space variables and with the seven point stencil finite differences method in three space variables. The right-hand side of the linear system derived from PDE (22)–(22.a) was computed as the product of the coefficient matrix A by the solution vector, with its components equal to unity.

4.1 Parallel modified generic factored approximate sparse inverse (PMGenFAspI) using AVX units

The performance, the speedup and the overall gain of the PMGenFAspI method parallelized for shared memory parallel systems with AVX units is examined for the 3D Poisson problem. The experimental results were obtained using an Intel Core-i7 4700MQ 2.4 GHz with 8 GB of RAM memory, running Windows 10 Pro.

Table 1 Performance and speedups of PMGenFAspI algorithm, using AVX units, for the 3D Poisson problem for various number of threads and values of the lfill parameter

Model problem	Threads	Performance			Speedups		
		lfill = 1	lfill = 2	lfill = 4	lfill = 1	lfill = 2	lfill = 4
3D Poisson $n = 1{,}000{,}000$	1	0.8436	1.4263	2.3273	–	–	–
	2	0.4644	0.7791	1.1775	1.8168	1.8308	1.9765
	4	0.2253	0.3898	0.6159	3.7439	3.6596	3.7789
	8	0.1871	0.3043	0.3363	4.5096	4.6877	6.9199
3D Poisson $n = 1{,}000{,}000$ AVX	1	0.2202	0.6090	1.6137	–	–	–
	2	0.1233	0.3405	0.8182	1.7867	1.7883	1.9722
	4	0.0849	0.2222	0.4931	2.5952	2.7412	3.2728
	8	0.0707	0.1772	0.2858	3.1164	3.4366	5.6456

Table 2 Overall gain of PMGenFAspI algorithm, using AVX units, for the 3D Poisson problem for various number of threads and values of the lfill parameter

Overall gain	Threads	lfill = 1	lfill = 2	lfill = 4
3D Poisson $n = 1{,}000{,}000$ AVX	1	3.8303	2.3423	1.4423
	2	6.8436	4.1887	2.8445
	4	9.9404	6.4206	4.7202
	8	11.9370	8.0494	8.1425

It should be noted that the software was developed in C++ without the use of scientific libraries. The software was compiled using the Visual Studio 2010 with OpenMP v2.0 and the maximum speed optimizations flag (/O2). The use of AVX units was realized through software libraries offered by the corresponding development environment.

In Table 1, the performance and speedup of the PMGenFAspI algorithm, using AVX units, for the 3D Poisson problem of order $n = 1{,}000{,}000$ for various number of threads and values of the lfill parameter are presented. In Table 2, the overall gain of the PMGenFAspI algorithm, using AVX units, for the 3D Poisson problem of order $n = 1{,}000{,}000$ for various number of threads and values of the lfill parameter is given.

It should be noted that the speedup of the PMGenFAspI method tends to the theoretical maximum as the number of threads is increased. The performance is increased using AVX units.

4.2 Parallel multilevel algebraic recursive generic approximate inverse solver

The performance of PMARGAIS is examined for solving various problems. Experimental results obtained from various systems and for various values of the parameters of the method are presented, to assess the behavior of the scheme. It should be stated that the following parameters were used for all the executions: dtol = 0.0, lfill =

Table 3 Performance and speedups of PMARGAIS method for the 2D Poisson problem for various values of the order (n), number of threads with Block size = 50, levels = 2 and the BBFS reordering scheme

	n	Performance			Speedups	
		Threads = 1	Threads = 2	Threads = 4	Threads = 2	Threads = 4
2D Poisson Bsize = 50 2 levels BBFS	10,000	0.2846	0.1729	0.1320	1.6460	2.1561
	90,000	2.8041	1.5797	0.9926	1.7751	2.8250
	250,000	8.6949	4.9598	3.1189	1.7531	2.7878
	490,000	19.3171	10.8979	6.6989	1.7726	2.8836
	810,000	35.8468	19.2370	11.6515	1.8634	3.0766
	1,000,000	46.0691	25.5934	15.3069	1.8000	3.0097
	1,440,000	69.7447	40.3776	24.6997	1.7273	2.8237
	1,960,000	106.0235	58.4938	35.0132	1.8126	3.0281
	2,560,000	144.4935	82.5649	52.2495	1.7501	2.7655
	2,890,000	172.8347	96.0503	60.1290	1.7994	2.8744

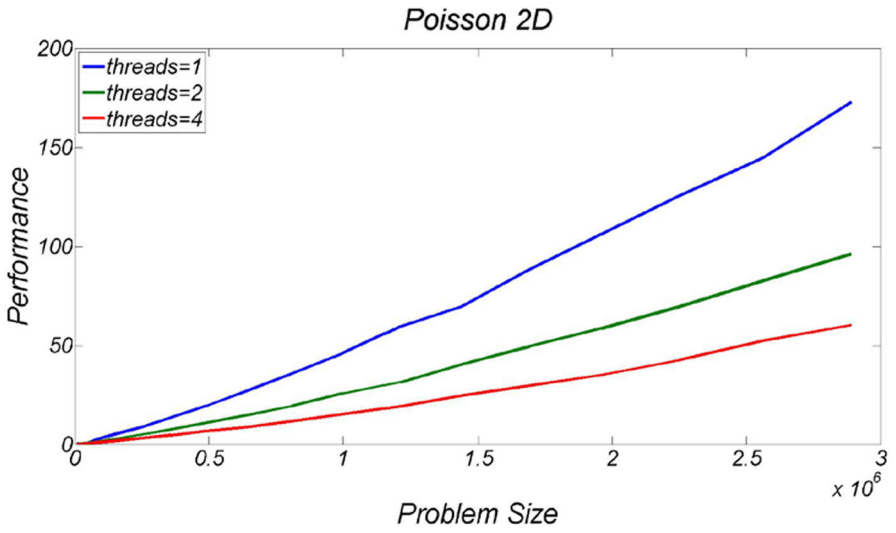

Fig. 3 Escalation of PMARGAIS method for the 2D Poisson problem for various values of the order (n) and number of threads with Block size = 50, levels = 2 and the BBFS reordering scheme

2, droptol = 0.1, ILUfill = 10 and ILUtol = 0.001, cf. [10]. The stopping criterion for the PMARGAIS method was $\|r_k\| < 10^{-10}\|r_0\|$, where r_i is the residual vector. The stopping criterion for the PEPBiCGSTAB method used in the last level was $\|r_k\| < 10^{-12}\|r_0\|$.

System 1 Numerical results were obtained using an AMD Phenom(tm) II X4 955 Processor 3.20 GHZ with 4 GB of RAM memory, running Ubuntu 12.04 LTS. It should be noted that the software was developed in C++ without the use of scientific libraries.

Table 4 Performance and speedups of PMARGAIS method for the 3D Poisson problem for various values of the order (n) and number of threads with Bsize = 1, levels = 2 and the MBBFS reordering scheme

	n	Performance			Speedups	
		Threads = 1	Threads = 2	Threads = 4	Thread = 2	Thread = 4
3D Poisson Bsize = 1 2 levels MBBFS	27,000	0.2127	0.1815	0.1695	1.1719	1.2549
	125,000	1.3624	1.0534	0.8442	1.2933	1.6138
	343,000	4.5452	3.2765	2.6030	1.3872	1.7461
	729,000	13.1469	9.0173	6.6356	1.4578	1.9813
	1,000,000	17.6166	12.1194	9.4103	1.4536	1.8721
	1,520,875	30.1416	19.7937	15.4303	1.5228	1.9534
	1,953,125	40.8718	28.1957	20.9582	1.4496	1.9502
	2,460,375	53.3979	36.3735	28.1247	1.4680	1.8986
	3,048,625	72.3477	49.2697	37.6895	1.4684	1.9196
	3,723,875	93.8577	61.2613	47.1386	1.5321	1.9911

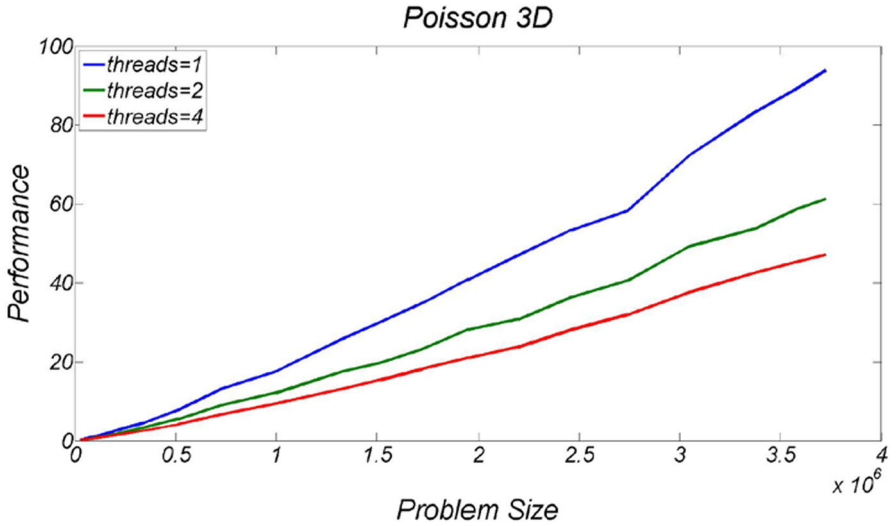

Fig. 4 Escalation of PMARGAIS method for the 3D Poisson problem for various values of the order (n) and number of threads with Block size = 50, levels = 2 and the BBFS reordering scheme

The software was compiled with g++ 4.6.3 with OpenMP v3.0 and the maximum optimizations flag (−O3).

In Table 3, the performance and speedup of PMARGAIS method for the 2D Poisson problem for various values of the order (n) and number of threads with Block size = 50, levels = 2 and the BBFS reordering scheme are given. In Fig. 3, the escalation of PMARGAIS method for the 2D Poisson problem for various values of the order (n) and the number of threads with Block size = 50, levels = 2 and the BBFS reordering scheme is depicted. In Table 4, the performance and speedups of PMARGAIS method

Table 5 Performance and speedups of PMARGAIS method for various problems, values of the Block size, number of threads and reordering schemes

Model problem	B size	Performance			Speedups	
		Threads = 1	Threads = 2	Threads = 4	Threads = 2	Threads = 4
tmt_unsym	15	190.8138	117.924	76.2462	1.6181	2.5026
$n = 917{,}825$	20	114.6582	66.6983	44.0186	1.7191	2.6048
3 levels	25	104.7484	64.0429	41.0811	1.6356	2.5498
BBFS	30	159.9528	98.6397	63.4345	1.6216	2.5215
cage13	1	9.5137	8.0920	7.0192	1.1757	1.3554
$n = 445{,}315$	2	12.406	10.0205	8.4776	1.2381	1.4634
2 levels	3	20.7175	16.732	13.8483	1.2382	1.4960
MBBFS	4	17.0560	13.5611	11.2221	1.2577	1.5196
	5	19.7831	15.8145	12.8718	1.2509	1.5369
	6	22.1682	17.4644	14.3224	1.2693	1.54780
	7	24.3199	19.3236	15.7927	1.2586	1.5399
	10	30.3313	24.0421	19.5113	1.2616	1.5546
cage13	1	9.5201	8.1271	7.0212	1.1714	1.3559
$n = 445{,}315$	2	18.2540	14.5264	12.1916	1.2566	1.4973
2 levels	3	20.8029	16.7427	13.8009	1.2425	1.5074
BBFS	4	29.5075	21.9046	17.9558	1.3471	1.6433
ATMOSMODD	1	41.9591	25.9568	19.5746	1.6165	2.1435
$n = 1{,}270{,}432$	10	64.1076	39.9283	28.1478	1.6056	2.2775
2 levels	15	62.5057	42.0861	26.5054	1.4852	2.3582
BBFS	20	65.0233	40.1462	28.7578	1.6197	2.2611
	25	68.7114	44.4020	28.9884	1.5475	2.3703
	30	76.8841	47.7144	33.6002	1.6113	2.2882
ATMOSMODD	1	42.1482	25.9399	19.3094	1.6248	2.1828
$n = 1{,}270{,}432$	10	65.3200	39.5534	28.0340	1.6514	2.3300
2 levels	15	64.4462	40.2910	27.9754	1.5995	2.3037
MBBFS	20	62.4860	41.3284	27.8987	1.5119	2.2397
	25	67.9544	42.7432	29.7789	1.5898	2.2820
	30	70.7867	43.3763	30.6878	1.6319	2.3067

for the 3D Poisson problem for various values of the order (n) and number of threads with Bsize = 1, levels = 2 and the MBBFS reordering scheme are presented. In Fig. 4, the escalation of PMARGAIS method for the 3D Poisson problem for various values of the order (n) and the number of threads with Block size = 50, levels = 2 and the BBFS reordering scheme is depicted. In Table 5, the performance and speedups of PMARGAIS method for various problems, values of the Block size, number of threads and reordering schemes are presented.

It can be easily seen that the speedups, presented in Tables 3 and 4, do not increase uniformly as the order (n) increases, which is due to the fact that the number of block

Table 6 Performance and speedups of PMARGAIS method for various problems, values of the Block size, number of levels, number of threads and reordering schemes

Model problem	B size	Performance				Speedups		
		Threads 1	Threads 2	Threads 4	Threads 8	Threads 2	Threads 4	Threads 8
2D Poisson $n = 1{,}000{,}000$ 2 levels BBFS	1	73.0583	41.7131	32.0228	24.9650	1.7514	2.2814	2.9264
	10	52.2918	27.5197	23.6618	18.2017	1.9002	2.2100	2.8729
	15	37.2686	22.3016	16.1839	13.7833	1.6711	2.3028	2.7039
	20	40.0517	24.7563	17.1673	14.8514	1.6178	2.3330	2.6968
	25	37.9516	21.4154	16.2204	13.1258	1.7722	2.3397	2.8914
	50	33.4608	19.0901	12.8044	10.8613	1.7528	2.6132	3.0807
	60	41.0946	22.8041	14.7068	12.6521	1.8021	2.7943	3.2481
2D Poisson $n = 1{,}000{,}000$ 3 levels BBFS	1	65.1896	40.8322	28.3215	22.0881	1.5965	2.3018	2.9513
	10	40.7832	24.6101	16.2110	13.1583	1.6572	2.5158	3.0994
	15	36.0839	20.6026	13.3689	11.0747	1.7514	2.6991	3.2582
	20	35.2060	20.1801	13.8575	10.9855	1.7446	2.5406	3.2048
	25	38.4092	20.2266	14.0181	11.3409	1.8989	2.7400	3.3868
	50	42.3599	23.6993	15.3683	12.6636	1.7874	2.7563	3.3450
	60	48.9286	27.4814	17.2894	14.7174	1.7804	2.8300	3.3245
3D Poisson $n = 1{,}000{,}000$ 2 levels MBBFS	1	12.3523	8.2733	6.6882	6.9908	1.4930	1.8469	1.7669
	2	29.6087	19.1731	15.8568	13.1233	1.5443	1.8673	2.2562
	5	35.4885	21.7388	16.7468	15.6908	1.6325	2.1191	2.2617
	10	40.8753	24.5271	20.1769	15.3356	1.6665	2.0259	2.6654
	20	43.0303	24.8755	18.4289	16.3608	1.7298	2.3349	2.6301

independent sets is not a multiple of the number of processors. Thus, the computational work is not balanced among processors and execution time of the parallel scheme is governed by the processor which is assigned the largest number of independent sets. The imbalance affects parallel performance significantly especially in the case where the block size has a large value and the number of independent sets is not a multiple of the number of processors.

System 2 Numerical results were obtained using an Intel Core-i7 4700MQ 2.4 GHz with 8 GB of RAM memory, running Windows 10 Pro.

It should be noted that the software was developed in C++ without the use of scientific libraries. The software was compiled using the Visual Studio 2010 with OpenMP v2.0 and the maximum speed optimizations flag (/O2). The use of AVX units was realized through software libraries offered by the corresponding development environment.

In Table 6, the performance and speedups of PMARGAIS method for various problems, values of the Block size, numbers of levels, number of threads and reordering schemes is given. In Table 7, the performance and speedup of PMARGAIS method, using AVX units, for various problems, values of the Block size and number of threads

Table 7 Performance and speedup of PMARGAIS method, using AVX units, for various problems, values of the Block size and number of threads with levels = 2 and MBBFS reordering scheme

Model problem	B size	Performance				Speedups		
		Threads 1	Threads 2	Threads 4	Threads 8	Threads 2	Threads 4	Threads 8
ATMOSMODD $n = 1{,}270{,}432$ 2 levels MBBFS	1	25.4312	15.5799	12.5642	11.8345	1.6323	2.0241	2.1489
	5	45.6125	28.7473	23.1224	21.0849	1.5867	1.9727	2.1633
	20	60.2285	35.3484	25.1605	20.8075	1.7039	2.3938	2.8946
	30	63.1542	37.3636	25.4148	22.0798	1.6903	2.4849	2.8603
	40	74.1339	41.9909	29.3320	24.1994	1.7655	2.5274	3.0635
ATMOSMODD $n = 1{,}270{,}432$ 2 levels MBBFS AVX	1	23.6567	14.7736	11.2694	10.3592	1.6013	2.0992	2.2836
	5	40.1180	25.6247	19.6499	18.3342	1.5656	2.0416	2.1882
	20	49.2225	29.5135	21.4837	18.2292	1.6678	2.2912	2.7002
	30	52.1364	31.9592	22.9823	20.1613	1.6313	2.2686	2.5860
	40	61.3299	35.7913	24.2216	22.5914	1.7135	2.5320	2.7147
cage13 $n = 445{,}315$ 2 levels MBBFS	1	8.6980	7.3315	6.7626	6.5930	1.1864	1.2862	1.3193
	5	13.6959	11.4181	9.9488	9.4726	1.1995	1.3766	1.4458
	20	32.8766	27.4312	23.6745	21.9267	1.1985	1.3887	1.4994
	30	57.9347	43.5103	39.6480	35.8269	1.3315	1.4612	1.6171
	40	216.7681	120.9654	70.9745	46.5487	1.7920	3.0542	4.6568
cage13 $n = 445{,}315$ 2 levels MBBFS AVX	1	7.2936	6.1280	5.6183	5.5628	1.1902	1.2982	1.3111
	5	12.5755	10.5042	8.7812	8.0882	1.1972	1.4321	1.5548
	20	29.5345	25.1745	21.6107	20.5358	1.1732	1.3667	1.4382
	30	50.6622	39.1946	36.3984	32.8196	1.2926	1.3919	1.5437
	40	198.1292	111.1254	65.1045	42.0849	1.7829	3.0432	4.7078

Table 8 Overall gain of PMARGAIS method, using AVX units, for various problems, values of the Block size and number of threads, with levels = 2 and the MBBFS reordering scheme

Overall gain	B size	Threads = 1	Threads = 2	Threads = 4	Threads = 8
ATMOSMODD $n = 1{,}270{,}432$ 2 levels MBBFS AVX	1	1.0750	1.7214	2.2567	2.4549
	5	1.1370	1.7800	2.3213	2.4878
	20	1.2236	2.0407	2.8035	3.3040
	30	1.2113	1.9761	2.7480	3.1324
	40	1.2088	2.0713	3.0606	3.2815
cage13 $n = 445{,}315$ 2 levels MBBFS AVX	1	1.1925	1.4194	1.5481	1.5636
	5	1.0891	1.3039	1.5597	1.6933
	20	1.1132	1.3060	1.5213	1.6009
	30	1.1435	1.4781	1.5917	1.7652
	40	1.0941	1.9507	3.3295	5.1507

with levels = 2 and MBBFS reordering scheme is presented. In Table 8, the overall gain of PMARGAIS method, using AVX units, for various problems, values of the Block size and number of threads with levels = 2 and MBBFS reordering scheme is given.

It should be mentioned that the use of the modified reordering scheme (MBBFS) instead of the BBFS scheme ensures load balancing and better control over the size of each block, especially for problems where the size of the blocks increases rapidly, such as matrices whose corresponding graph contains multiple vertices of large degree. Hence, the MBBFS method reduces the memory requirements for storing the block diagonal inverse, giving an upper limit of the memory requirements of the method.

It should be also noted that the results can be further improved by modifying the MBBFS method to produce blocks that their number is equal to a multiple of the processor units.

5 Conclusion

The proposed parallel multilevel algebraic recursive generic approximate inverse solver (PMARGAIS) is efficient for solving a class of problems resulting in large sparse linear systems. The hybrid direct-iterative PMARGAIS method involves dense computations that can be parallelized efficiently, using AVX units. Dense computations are more efficiently parallelized than sparse computations leading to increased performance and better scalability. Moreover, the proposed scheme is based on the modified reordering scheme (MBBFS) retaining balanced computational work, thus enhancing performance and corresponding speedups are close to theoretical maximum. The PMARGAIS method is based on the PBiCGSTAB method in conjunction with PMGenFAspI matrix, using AVX units. The use of AVX units in conjunction with multicore systems for computing the PMGenFAspI matrix results in increased speedups that surpass the number of available processors. Moreover, the PBiCGSTAB has been parallelized for multicore systems with AVX units resulting in improved performance. Future work will be focused on the implementation of the method on hybrid distributed memory systems.

Appendix

The PEPBiCGSTAB method, using AVX units, is described by the following algorithmic scheme:

Parallel Explicit Preconditioned Bi-Conjugate Gradient STABilized Algorithm (PEPBiCGSTAB) with AVX

Let x_0 be an arbitrary initial approximation to the solution vector x and r_0 the residual vector for the initial approximation. Then,

r_0=b-A*x_0; $r_0^{'}$=r_0; ρ_0=α=ω_0=1; v_0=p_0=0

Then, for i=1,...,(until convergence) compute the vectors x_i, r_i, z_i, y_i, p_i, s_i, t_i **and the scalar quantities** α, β, ω_i, ρ_i **as follows:**

Parallel For j=1,...,m

Load 4 double numbers from $r_0^{'}$ **in main memory to register** *xr1*

Load 4 double numbers from r_{i-1} **in main memory to register** *xr2*

xr3=fmadd(*xr3,xr1,xr2*)

End For (j)

$xr3=xr3_{proc(1)}+xr3_{proc(2)}+...xr3_{proc(k)}$; ρ_i=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)

Add remaining products $r_0^{'}(j)*r_{i-1}(j)$ **to** ρ_i

$\beta=(\rho_i/\rho_{i-1})/(\alpha/\omega_{i-1})$

Parallel For j=1,...,m **with step** 4

Load 4 times the double number ω_{i-1} **to register** *xr1*

Load 4 double numbers from v_{i-1} **in main memory to register** *xr2*

Load 4 double numbers from p_{i-1} **in main memory to register** *xr3*

Load 4 times the double number β **to register** *xr4*

Load 4 double numbers from r_{i-1} **in main memory to register** *xr5*

xr6= xr5+xr4(xr3-xr1*xr2)*

Store the 4 double numbers from register *xr6* **to** p_i

End For (j)

Compute remaining $p_i(j)=r_{i-1}(j)+\beta(p_{i-1}(j)-\omega_{i-1}v_{i-1}(j))$

Parallel For j=1,...,m

For k∈M(j,:) **with step** 4

Load 4 double numbers from M(j,:) **in main memory to register** *xr1*

Load 4 double numbers from p_i **in main memory to register** *xr2*

xr3=fmadd(*xr3,xr1,xr2*)

End For (k)

$y_i(j)$=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)

Add remaining products M(j,k)*p_i(k) **to** $y_i(j)$

End For (j)

Parallel For j=1,...,m

For k∈A(j,:) **with step** 4

Load 4 double numbers from A(j,:) **in main memory to register** *xr1*

Load 4 double numbers from y_i **in main memory to register** *xr2*

xr3=fmadd(*xr3,xr1,xr2*)

End For (k)

$v_i(j)$=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)

Add remaining products A(j,k)*y_i(k) **to** $v_i(j)$

End For (j)

Parallel For j=1,...,m

Load 4 double numbers from $r_0^{'}$ **in main memory to register** *xr1*

Load 4 double numbers from r_{i-1} **in main memory to register** *xr2*

xr3=fmadd(*xr3,xr1,xr2*)

End For (j)

$xr3=xr3_{proc(1)}+xr3_{proc(2)}+...xr3_{proc(k)}$; sum=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)

Add remaining products $r_0^{'}(j)*r_{i-1}(j)$ **to** sum

$\alpha=\rho_i$/sum

Parallel For j=1,...,m **with step** 4
  Load 4 times the double number a **to register** *xr1*
  Load 4 double numbers from v_i **in main memory to register** *xr2*
  Load 4 double numbers from r_{i-1} **in main memory to register** *xr3*
  *xr4=xr3-xr1*xr2*
  Store the 4 double numbers from register *xr4* **to** s_i
End For (j)
Compute remaining $s_i(j)=r_{i-1}(j)-\alpha*v_i(j)$
Parallel For j=1,...,m
  For k∈M(j,:) **with step** 4
    Load 4 double numbers from M(j,:) **in main memory to register** *xr1*
    Load 4 double numbers from s_i **in main memory to register** *xr2*
    xr3=fmadd(*xr3*,*xr1*,*xr2*)
  End For (k)
  $z_i(j)$=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)
  Add remaining products M(j,k)*s_i(k) **to** z_i(j)
  End For (j)
  Parallel For j=1,...,m
    For k∈A(j,:) **with step** 4
      Load 4 double numbers from A(j,:) **in main memory to register** *xr1*
      Load 4 double numbers from z_i **in main memory to register** *xr2*
      xr3=fmadd(*xr3*,*xr1*,*xr2*)
    End For (k)
    $t_i(j)$=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4)
    Add remaining products A(j,k)*z_i(k) **to** t_i(j)
  End For (j)
  Parallel For j=1,...,m
    Load 4 double numbers from t_i **in main memory to register** *xr1*
    Load 4 double numbers from s_i **in main memory to register** *xr2*
    xr3=fmadd(*xr3*,*xr1*,*xr2*); *xr4*=fmadd(*xr4*,*xr1*,*xr1*)
  End For (j)
  $xr3=xr3_{proc(1)}+xr3_{proc(2)}+...xr3_{proc(k)}$; $xr4=xr4_{proc(1)}+xr4_{proc(2)}+...xr4_{proc(k)}$
  sum=*xr3*(1)+*xr3*(2)+*xr3*(3)+*xr3*(4); sum2=*xr4*(1)+*xr4*(2)+*xr4*(3)+*xr4*(4)
  Add remaining products $t_i(j)*s_i(j)$ **to** sum
  Add remaining products $t_i(j)*t_i(j)$ **to** sum2
  ω_i=sum/sum2
  Parallel For j=1,...,m **with step** 4
    Load 4 times the double number ω_i **to register** *xr1*
    Load 4 double numbers from z_i **in main memory to register** *xr2*
    Load 4 times the double number α **to register** *xr3*
    Load 4 double numbers from y_i **in main memory to register** *xr4*
    Load 4 double numbers from x_{i-1} **in main memory to register** *xr5*
    *xr6=xr5+xr3*xr4+xr1*xr2*
    Store the 4 double numbers from register *xr6* **to** x_i
  End For (j)
  Compute remaining $x_i(j)=x_{i-1}(j)+\alpha*y_i(j)+\omega_i z_i(j)$
  Parallel For j=1,...,m **with step** 4
    Load 4 times the double number ω_i **to register** *xr1*
    Load 4 double numbers from t_i **in main memory to register** *xr2*
    Load 4 double numbers from s_i **in main memory to register** *xr3*
    *xr4=xr3-xr1*xr2*
    Store the 4 double numbers from register *xr4* **to** r_i
  End For (j)
  Compute remaining $r_i(j)=s_i(j)-\omega_i*t_i(j)$
End For (i)

where $fmadd(xr3, xr1, xr2)$ is the fused multiply add operation $xr3 = xr3 + xr1 * xr2$, where $xr1$, $xr2$ and $xr3$ are vectors consisting of four double-precision floating point numbers.

References

1. Arabnia HR, Thapliyal H, Vinod AP (2006) Combined integer and floating point multiplication architecture (CIFM) for FPGAs and its reversible logic implementation. In: 49th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS'06), San Juan, Puerto Rico, August 6–9, pp 148–154
2. Axelsson O (1996) Iterative solution methods. Cambridge University Press, Cambridge
3. Benzi M, Meyer CD, Tuma M (1996) A sparse approximate inverse preconditioner for the conjugate gradient method. SIAM J Sci Comput 17(5):1135–1149
4. Botta EFF, van der Ploeg A, Wubs FW (1996) Nested grids ILU- decomposition (NGILU). J Comp Appl Math 66:515–526
5. Botta EFF, Wubs W (1997) MRILU: it's the preconditioning that counts. Technical Report W-9703, Department of Mathematics, University of Groningen, The Netherlands
6. Chapman B, Jost G, Van Der Pas R (2008) Using OpenMP: portable shared memory parallel programming. The MIT Press, Cambridge
7. Chow E (2001) Parallel implementation and practical use of sparse approximate inverses with a priori sparsity patterns. Int J High Perf Comput Appl 15:56–74
8. Chow E (2000) A priori sparsity patterns for parallel sparse approximate inverse preconditioners. SIAM J Sci Comput 21:1804–1822
9. Davis TA, Hu Y (2011) The University of Florida sparse matrix collection. ACM Trans Math Softw (TOMS) 38(1):1–25
10. Filelis-Papadopoulos CK, Gravvanis GA (2016) A class of generic factored and multilevel recursive approximate inverse techniques for solving general sparse systems. Eng Comp 33(1):74–99
11. Golub GH, Reinsch C (1970) Singular value decomposition and least squares solutions. In: Wilkinson JH, Reinsch C (eds) Handbook for automatic computation, vol. 2 (Linear Algebra). Springer-Verlag, New York, pp 134–151
12. Golub GH, Van Loan CF (1996) Matrix computations, 3rd edn. Johns Hopkins University Press, Baltimore
13. Gravvanis GA (2009) High performance inverse preconditioning. Arch Comput Meth Engin 16(1):77–108
14. Gravvanis GA, Filelis-Papadopoulos CK, Matskanidis PI (2014) Algebraic multigrid methods based on generic approximate banded inverse matrix techniques. Comput Model Eng Sci (CMES) 100(4):323–345
15. Grote MJ, Huckle T (1997) Parallel preconditioning with sparse approximate inverses. SIAM J Sci Comput 18(3):838–853
16. Grote MJ, Huckle T (1995) Effective parallel preconditioning with sparse approximate inverses. In: Proceedings of SIAM Conference on Parallel Processing for Scientific Computing, SIAM, pp 466–471
17. Intel Volume 1. Basic Architecture: http://www.c-jump.com/CIS77/reference/Intel/CIS77_24319002/index.html
18. Kolotolina YuL, Yeremin YuA (1993) Factorized sparse approximate inverse preconditionings. I. Theory. SIAM J Matrix Anal Appl 14:45–58
19. Manguoglu M (2011) A domain-decomposing parallel sparse linear system solver. J Comput Appl Math 236(3):319–325
20. Meijerink JA, Van der Vorst HA (1977) An iterative method for linear systems of which the coefficient is a symmetric M-matrix. Math Comput 31:148–162
21. Ruge A, Stuben K (1987) Algebraic multigrid. In: McCormick (ed) Multigrid methods. Front Appl Math 3(4) SIAM
22. Saad Y (1994) ILUT: a dual threshold incomplete LU factorization. Num Linear Algebra Appl 1(4):387–402
23. Saad Y, Suchomel B (2002) ARMS: an algebraic recursive multilevel solver for general sparse linear systems. Num Linear Algebra Appl 9(5):359–378

24. Saad Y, Zhang J (1999) BILUTM: a domain-based multilevel block ILUT preconditioner for general sparse matrices. SIAM J Matrix Anal Appl 21:279–299
25. Thapliyal H, Arabnia HR (2006) Reversible programmable logic array (RPLA) using Fredkin and Feynman gates for industrial electronics and applications. In: Proceedings of the International Conference on Computer Design and Conference on Computing in Nanotechnology (CDES'06), Las Vegas, USA, June 26–29, ISBN #: 1-60132-009-4. http://arxiv.org/abs/cs/0609029, pp 70–74
26. Thapliyal H, Srinivas MB, Arabnia HR (2005) Reversible logic synthesis of half, full and parallel subtractors. In: Proceedings of the International Conference on Embedded Systems and Applications, ESA'05, June, Las Vegas, pp 165–181
27. Trottenberg U, Osterlee CW, Schuller A (2000) Multigrid. Academic Press, Cambridge
28. Van der Vorst HA (1992) Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. SIAM J Sci Stat Comput 13(2):631–644