Comput Mech
DOI 10.1007/s00466-014-1084-3

ORIGINAL PAPER

Impact of data distribution on the parallel performance of iterative linear solvers with emphasis on CFD of incompressible flows

M. Esmaily-Moghadam · Y. Bazilevs · A. L. Marsden

Received: 28 July 2014 / Accepted: 23 September 2014

Abstract A parallel data structure that gives optimized memory layout for problems involving iterative solution of sparse linear systems is developed, and its efficient implementation is presented. The proposed method assigns a processor to a problem subdomain, and sorts data based on the shared entries with the adjacent subdomains. Matrix–vector-product communication overhead is reduced and parallel scalability is improved by overlapping inter-processor communications and local computations. The proposed method simplifies the implementation of parallel iterative linear equation solver algorithms and reduces the computational cost of vector inner products and matrix–vector products. Numerical results demonstrate very good performance of the proposed technique.

Keywords Finite element · Linear solver · Parallel data structure

1 Introduction

Large systems of linear equations arise in many areas of computational science and engineering. These linear systems, which are often a byproduct of a finite-difference, finite-volume, or finite-element discretization of partial differential equations (PDEs), are solved using a linear equation solver (LS). In general, the LS contributes a large portion of the total computational cost of a PDE solver, especially in applications such as computational fluid dynamics (CFD). Hence, improving the LS, thereby reducing the cost of these simulations, is an active area of research [7,21–26,28,29,33–36,38–40,43]. These efforts are mainly focused on iterative rather than direct solvers for large-scale time-dependent simulations, mainly because of their modest memory requirements and good parallel scalability.

Parallel processing is utilized to offset the high computational cost of large-scale PDE-based simulations. Designing efficient parallel algorithms, however, remains a challenge and continues to attract significant attention in the community [4,5,17,19,20,27,30,32,42]. In parallel PDE-based simulations, the inter-processor communications, which takes a significant portion of the total computational time, are required to carry out the computations. The inter-processor communications are generally required inside the LS, where the entire discrete solution, although partitioned between the processors, is calculated simultaneously using global operations. The high cost of global operations, either collective or processor-to-processor, is partly caused by waiting before sending or receiving messages. In larger problems, this wait time can contribute to a larger portion of the overall wall-clock time since a larger number of processors are employed. Hence, maintaining scalability and improving parallel efficiency at higher number of processors remains a challenge.

In this paper we present an algorithm for efficient handling of parallel algebraic operations in iterative solvers. This algorithm is based on a *low-entropy data structure* that maps the sparse matrix row and column indices on each processor to reduce the wait time by overlapping communication and computation, which is key for improving parallel efficiency. Although preliminary results for the proposed tech-

M. Esmaily-Moghadam · A. L. Marsden
Department of Mechanical and Aerospace Engineering, University of California, San Diego, USA
e-mail: mesmaily@ucsd.edu

Y. Bazilevs (✉)
Department of Structural Engineering, University of California, San Diego, USA
e-mail: yuri@ucsd.edu

Published online: 25 October 2014

nique were previously reported in a short conference proceedings paper [9], here the method is presented in full detail, with additional simplifications and improvements, and with a more complete set of numerical results.

The paper is outlined as follows. In Sect. 2, after briefly recalling the basic concepts, we present and explain an algorithm to produce an appropriately sorted node list. We then show how to use the sorted node list to perform algebraic operations commonly used in iterative linear solvers: matrix–vector product and vector inner product. In Sect. 3, we test the method using three examples. The advantages of using the low-entropy data structure are clearly demonstrated using a wide range of mesh sizes and processor numbers. The performance of the proposed algorithm is also tested on a patient-specific blood flow CFD example, and scalability and CPU times of the computations are reported. In Sect. 4 we draw conclusions. In the Appendix we report the performance comparison of the proposed algorithm with that of PETSc [1], a well-known, open source library of linear-algebra routines.

2 Sorted data structure

In this section, we discuss a general approach for performing basic algebraic operations in parallel, present the data sorting algorithm, and describe its implementation. Assuming that the underlying PDE is scalar-valued, we associate the unknowns in the linear system of equations with the mesh nodes. As a result, the terms *node* and *entry* may used interchangeably. This is done for simplicity of exposition and is not a requirement of the proposed method. In what follows, roman subscripts are used to denote variable names and italic superscripts are used as indices. Subscripts g and s are generally used to denote the global and sorted counterparts, respectively. Indices i, j, and k are used for processor IDs, and p and q for node IDs. A pure message passing interface (MPI) programming paradigm is assumed in the developments that follow.

2.1 Basic concepts

The first step in solving a PDE in parallel is to partition the mesh and assign each processor to a partition (i.e., a subdomain). As a result, in what follows, the terms *processor ID* and *partition number* will have the same meaning. There are two choices for mesh partitioning, namely, node-based (also called vertex-based) and element-based, in which elements and nodes are shared between adjacent processors, respectively.

We adopt the element-based partitioning approach. The physical domain $\Omega \subset \mathbb{R}^{\text{nsd}}$ is partitioned into a set of non-overlapping subdomains Ω^i as

$$\bigcup_{i=1}^{n_\text{p}} \Omega^i = \Omega, \tag{1}$$

$$\Omega^i \cap \Omega^j = \Gamma^{ij}, \tag{2}$$

where n_p is the number of subdomains assumed to be equal to the number of MPI processors, and Γ^{ij}, an nsd $-$ 1-dimensional manifold in $\mathbb{R}^{\text{nsd}}$, is the boundary between the subdomains i and j. In this case, the information corresponding to nodes on Γ^{ij} is shared between the processors.

We denote a linear system of equations by

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3}$$

where $\boldsymbol{A}$ is an $n \times n$ left-hand-side matrix, $\boldsymbol{b}$ is an $n \times 1$ right-hand-side vector, and $\boldsymbol{x}$ is an $n \times 1$ solution vector.

After assembling the matrix and vector regardless of the neighboring subdomains, each processor contains a part of $\boldsymbol{A}$ and $\boldsymbol{b}$,

$$\sum_{i=1}^{n_\text{p}} \boldsymbol{A}_\text{g}^i = \boldsymbol{A}, \tag{4}$$

$$\sum_{i=1}^{n_\text{p}} \boldsymbol{b}_\text{g}^i = \boldsymbol{b}, \tag{5}$$

where $\boldsymbol{A}_\text{g}^i$ and $\boldsymbol{b}_\text{g}^i$ are the global representations of the contributions of Ω^i to the left-hand-side matrix and right-hand-side vector, respectively.

Denoting the number of non-zero entries in $\boldsymbol{b}_\text{g}^i$ by n^i, the non-zero entries of $\boldsymbol{b}_\text{g}^i$ can be mapped to a local vector, $\boldsymbol{b}^i$, with size n^i using a permutation matrix $\boldsymbol{P}^i$ as

$$\boldsymbol{b}^i = \boldsymbol{P}^i \boldsymbol{b}_\text{g}^i. \tag{6}$$

The transpose of the permutation matrix may be written as

$$\boldsymbol{P}^{i\,T} = \left[\mathbf{1}^{a^i(1)}\ \mathbf{1}^{a^i(2)}\ \cdots\ \mathbf{1}^{a^i(n^i)}\right], \tag{7}$$

where $\mathbf{1}^p$ is a vector with a unit entry at location p, i.e.,

$$1^p(q) = \begin{cases} 0 & q \neq p \\ 1 & q = p, \end{cases} \tag{8}$$

and $\boldsymbol{a}^i$ is the unsorted list of all nodes in subdomain Ω^i. Similarly, the left-hand-side may be written as

$$\boldsymbol{A}^i = \boldsymbol{P}^i \boldsymbol{A}_\text{g}^i \boldsymbol{P}^{i\,T}, \tag{9}$$

where $\boldsymbol{A}^i$ is the local counterpart of $\boldsymbol{A}_\text{g}^i$.

In general, the distribution of non-zero entries in $\boldsymbol{b}_\text{g}^i$, or a particular column or row of $\boldsymbol{A}_\text{g}^i$ does not follow a specific rule.

Hence, the shared entries between $\boldsymbol{b}^i$ and $\boldsymbol{b}^j$ are randomly distributed among the unshared entries. From the point of view of inter-processor communication this randomness has some drawbacks. A random distribution requires random fetches from the main memory during the communications, which reduces the cache hit ratio. Furthermore, shared entries need special handling that largely depends on the nature of the algebraic operations performed. In what follows, we present a local-to-global node mapping that removes the above mentioned drawbacks, and leads to a reduced communication overhead.

2.2 Local data representation

The most common algebraic operations that take place in iterative linear algebra solvers can be divided into three groups:

(1) Addition and scaling of vectors. These operations are simple and produce mathematically consistent results regardless the node ordering.
(2) Calculation of vector inner products and norms. In this group of operations special care must be taken when handling the shared entries of $\boldsymbol{b}^i$ and $\boldsymbol{b}^j$. A traditional approach is to send the shared entries from processor j to processor i, set the shared entries on j to zero, perform the operation on the vectors local to the processors, and sum the values on all processors to obtain the final result.
(3) Calculation of matrix–vector products. Special care is also required here since the shared rows in $\boldsymbol{A}^i$ only contain a part of the corresponding rows in $\boldsymbol{A}$, which is a consequence of element-based partitioning. Traditionally, the product $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ is calculated in parallel using four steps: First, it is ensured that shared entries of $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$ contain values equal to the corresponding entries of $\boldsymbol{x}$; Second, the local-to-processor matrix–vector products are performed; Third, the shared entries of $\boldsymbol{y}^i$ and $\boldsymbol{y}^j$ are added; Finally, the results are "scattered" back to processors i and j.

Remark The vector norm considered here is a standard Euclidean norm.

To obtain a more favorable substrate for the second and third group of operations, we start from the list of global node IDs $\boldsymbol{a}^i$, and modify it to produce a sorted list $\boldsymbol{a}^i_{\mathrm{s}}$ using the mapping $\boldsymbol{\mathcal{M}}^i$, where

$$p_{\mathrm{s}} = \mathcal{M}^i(p), \tag{10}$$

$$a^i_{\mathrm{s}}(p_{\mathrm{s}}) = a^i(p), \tag{11}$$

and $p, p_{\mathrm{s}} \in \{1, \ldots, n^i\}$. In practice, $\boldsymbol{\mathcal{M}}^i$ is a pointer array with length n^i that orders the local representation of data. The mapping $\boldsymbol{\mathcal{M}}^i$ is found, such that, given n^i and $\boldsymbol{a}^i$, the following holds:

(1) The *owner* of a node p shared between m processors $\{i^1, \ldots, i^m\}$ is i^m, given $i^m > i^j \ \forall j \in \{1, \ldots, m-1\}$. In this case, processor i^m is called the owner of p and $i^j \ \forall j \neq m$ will only keep a copy of node p value.
(2) The conditions

$$\left.\begin{array}{c} a^i_{\mathrm{s}}(p) \in \boldsymbol{a}^j_{\mathrm{s}} \\ a^i_{\mathrm{s}}(q) \notin \boldsymbol{a}^k_{\mathrm{s}} \ \forall k \neq i \\ i < j \end{array}\right\} \Rightarrow q < p, \tag{12}$$

are satisfied. In this case p is possibly owned by j, and q is "only" owned by i.
(3) The conditions

$$\left.\begin{array}{c} a^i_{\mathrm{s}}(p) \in \boldsymbol{a}^j_{\mathrm{s}} \\ a^i_{\mathrm{s}}(q) \notin \boldsymbol{a}^k_{\mathrm{s}} \ \forall k \neq i \\ j < i \end{array}\right\} \Rightarrow p < q, \tag{13}$$

are satisfied similar to those in Eq. (12), which ensure that entries shared by the lower- and higher-numbered processors are located in the beginning and end of the vector, respectively. Note that both p and q are owned by i in Eq. (13).

Equation (12) presents a more relaxed set of conditions compared to those presented in [9], which are repeated here for convenience:

$$\left.\begin{array}{c} a^i_{\mathrm{s}}(p) \in \boldsymbol{a}^j_{\mathrm{s}} \\ a^i_{\mathrm{s}}(q) \in \boldsymbol{a}^k_{\mathrm{s}} \\ j < k \end{array}\right\} \Rightarrow p < q. \tag{14}$$

Equation (14) ensures a fully sorted vector from low to high processor ID on any processor i, while Eqs. (12) and (13) only mandate unowned entries to be located at the end, and owned, shared entries at the beginning of the vector (see Fig. 1 for an illustration). This simplifies the computation of $\boldsymbol{\mathcal{M}}$, specifically in cases that an entry is shared between more than two processors, hence eliminating the need to track these entries and to use an extra hash table.

We denote the last owned shared entry by n^i_{s} and the last owned unshared entry by n^i_{o}, i.e.,

$$n^i_{\mathrm{s}} = \sup\left\{p \in \{1, \ldots, n^i\} : a^i_{\mathrm{s}}(p) \in \boldsymbol{a}^j_{\mathrm{s}}, j < i\right\}, \tag{15}$$

and

$$n^i_{\mathrm{o}} = \sup\left\{p \in \{1, \ldots, n^i\} : a^i_{\mathrm{s}}(p) \notin \boldsymbol{a}^j_{\mathrm{s}}, \forall j \neq i\right\}. \tag{16}$$

Note that

$$n^i_{\mathrm{s}} = \inf\left\{p \in \{1, \ldots, n^i\} : a^i_{\mathrm{s}}(p) \notin \boldsymbol{a}^j_{\mathrm{s}}, \forall j \neq i\right\} - 1, \tag{17}$$

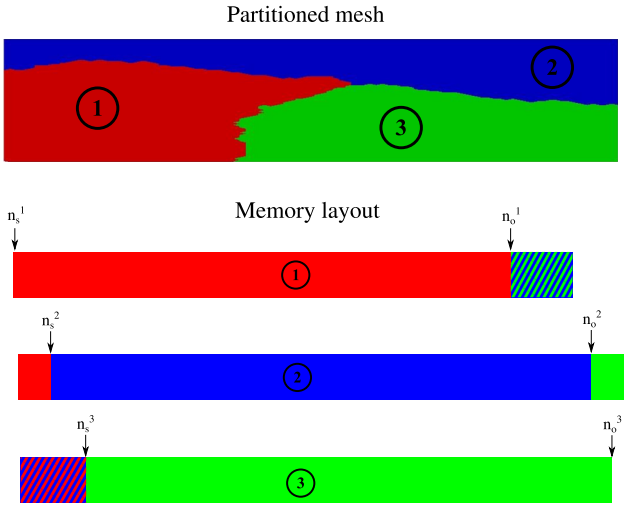

Fig. 1 Data distribution in memory for a partitioned mesh with 3 processors. Encircled numbers denote processor IDs. The initial and final segments of vectors are shared by lower and higher processors, respectively. Hatched areas with multiple *colors* show segments of vectors that contain entries shared with processors with corresponding colors

and

$$n_{\mathrm{o}}^i = \inf\left\{p \in \{1,\ldots,n^i\} : a_{\mathrm{s}}^i(p) \in \boldsymbol{a}_{\mathrm{s}}^j, i < j\right\} - 1. \quad (18)$$

(See Fig. 1 for an illustration.)

Given $\boldsymbol{a}^i$, n^i, n, and n_{p}, we obtain $\boldsymbol{\mathcal{M}}^i$ as follows:

(1) Construct a global-to-local array $\boldsymbol{a}_{\mathrm{g}}^i$ with size n that is initialized to zero and communicate $\{\boldsymbol{a}^1,\ldots,\boldsymbol{a}^{n_{\mathrm{p}}}\}$ between all processors to construct an $\max(n^i) \times n_{\mathrm{p}}$-dimensional array $\boldsymbol{d}^i$ as:

```
a_g^i ← 0
do p = 1, ..., n^i
  a_g^i(a^i(p)) ← p
  do j = 1, ..., n_p
    d^i(p, j) ← a^j(p)
```

(2) Construct the shared segments of $\boldsymbol{a}_{\mathrm{s}}^i$ as:

```
n_s^i ← 0
n_o^i ← n^i
do j = n_p, ..., 1
  if j ≠ i
    do p = 1, ..., n^j
      p_g ← d^i(p, j)
      q ← a_g^i(p_g)
      if q ≠ 0
        if d^i(q, i) ≠ 0
          d^i(q, i) ← 0
          if j < i
            n_s^i ← n_s^i + 1
            a_s^i(n_s^i) ← p_g
          else
            a_s^i(n_o^i) ← p_g
            n_o^i ← n_o^i − 1
```

Note the third *if* statement and the assignment statement that follows it, where $\boldsymbol{d}^i(:,i)$ is used to make sure that the nodes are included in $\boldsymbol{a}_{\mathrm{s}}^i$ only once.

(3) Construct the rest of $\boldsymbol{a}_{\mathrm{s}}^i$, the segment that contains nodes only owned by i, as:

```
q ← n_s^i + 1
do p = 1, ..., n^i
  p_g ← d^i(p, i)
  if p_g ≠ 0
    a_s^i(q) ← p_g
    q ← q + 1
```

(4) Reconstruct the global-to-local array $\boldsymbol{a}_{\mathrm{g}}^i$ based on the sorted list $\boldsymbol{a}_{\mathrm{s}}^i$, and calculate the mapping array as:

```
do p = 1, ..., n^i
  a_g^i(a_s^i(p)) ← p

do p = 1, ..., n^i
  M^i(p) ← a_g^i(a^i(p))
```

Remarks

(1) The above algorithm assumes that all arrays start from index 1, which is the Fortran default.

(2) As superscript i suggests, all arrays are processor-specific, and calculations are performed independently on all processors.

(3) The proposed method is not memory intensive. The largest arrays employed, $\boldsymbol{a}_{\mathrm{g}}^i$ and $\boldsymbol{d}^i$, are approximately n bytes each. This value is independent of the number of processors, hence the memory requirements do not increase with n_{p}.

(4) The number of operations is also proportional to n on each processor, therefore the wall-clock time (or elapsed time) for the calculations is independent of n_{p} as well. This is confirmed by the test cases reported in the numerical results section.

We present a simple example to illustrate the node numbering scheme and the data structures employed. We consider a 2D four-element and nine-node mesh decomposed into three processors, with $n^1 = 6$, $n^2 = 4$, and $n^3 = 4$ (see Fig. 2 for an illustration). Assuming an unsorted list of nodes for each partition,

$$\boldsymbol{a}^1 = \begin{bmatrix} 7 \\ 3 \\ 4 \\ 9 \\ 8 \\ 1 \end{bmatrix}, \quad \boldsymbol{a}^2 = \begin{bmatrix} 2 \\ 3 \\ 6 \\ 9 \end{bmatrix}, \quad \boldsymbol{a}^3 = \begin{bmatrix} 1 \\ 9 \\ 2 \\ 5 \end{bmatrix}, \quad (19)$$

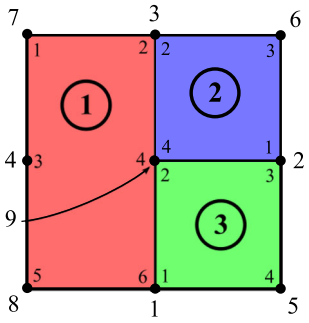

Fig. 2 An example of a mesh with 9 nodes partitioned to three subdomains that are denoted by encircled numbers. The global and local node IDs are shown using the *numbers* outside and inside of the *box*, respectively. For example, node 9 (central node) is local node 2 on processor 3 and 4 on processors 1 and 2

we use the above algorithm to obtain

$$\boldsymbol{a}_{\mathrm{s}}^1 = \begin{bmatrix} 7 \\ 4 \\ 8 \\ 3 \\ 9 \\ 1 \end{bmatrix}, \quad \boldsymbol{a}_{\mathrm{s}}^2 = \begin{bmatrix} 3 \\ 6 \\ 2 \\ 9 \end{bmatrix}, \quad \boldsymbol{a}_{\mathrm{s}}^3 = \begin{bmatrix} 2 \\ 9 \\ 1 \\ 5 \end{bmatrix}, \tag{20}$$

and

$$\mathcal{M}^1 = \begin{bmatrix} 1 \\ 4 \\ 2 \\ 5 \\ 3 \\ 6 \end{bmatrix}, \quad \mathcal{M}^2 = \begin{bmatrix} 3 \\ 1 \\ 2 \\ 4 \end{bmatrix}, \quad \mathcal{M}^3 = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 4 \end{bmatrix}. \tag{21}$$

In this example, $n_{\mathrm{s}}^1 = 0$, $n_{\mathrm{o}}^1 = 3$, $n_{\mathrm{s}}^2 = 1$, $n_{\mathrm{o}}^2 = 2$, $n_{\mathrm{s}}^3 = 3$, and $n_{\mathrm{o}}^3 = 4$. There are two shared nodes between any two processors, hence two blocks of data with length two are sent and received in a communication after a matrix–vector product. More discussion is given in the what follows.

2.2.1 *Benefits for vector inner-product and norm computations*

In the computation of a vector norm that is mapped by $\mathcal{M}^i$, only locally owned entries are considered. In practice this can be achieved by simply setting the upper limit of the nested loop to n_{o}^i instead of n^i. This is a valid approach since all the entries following n_{o}^i are owned by other processors and will be included in the calculations exactly once. As a result, the entries of $\boldsymbol{b}^i(p)$, $p \in \{n_{\mathrm{o}}^i + 1, \ldots, n^i\}$ do not have any effect on the norm or dot product calculation and may retain any value. This is in contrast to the traditional method that requires setting those entries to zero to avoid computing a single entry twice. The proposed renumbering reduces extra computations associated with the assignment of the entries $n_{\mathrm{o}}^i + 1$ to n^i to zero and looping over them. Also, depending on how an iterative linear solver is implemented, since shared-unowned entries are not changed, the proposed mapping can reduce the amount of data communicated between the processors. Note that $\sum_{i=1}^{n_{\mathrm{p}}} n_{\mathrm{o}}^i = n$ is the number of nodes participating in the inner-product computation. It is coincident with the total number of nodes in the mesh, and is independent of the number of mesh partitions n_{p}.

2.2.2 *Benefits for matrix–vector product computations*

Using the sorted list of nodes is also beneficial in matrix–vector products. The benefits are achieved by using non-blocking communications calls overlapped with computations. The parallel procedure involves the following steps:

(1) Perform the matrix-vector products at the processor level for the shared rows, from 1 to n_{s}^i and from $n_{\mathrm{o}}^i + 1$ to n^i.
(2) Wait for the send requests.
(3) Copy the resulting values in the shared entries to the buffer, and call non-blocking processor-to-processor send and receive routines.
(4) While the messages are being delivered, calculate the unshared rows, from $n_{\mathrm{s}}^i + 1$ to n_{o}^i.
(5) Wait for the receive requests.
(6) Add the received values to the corresponding vector entries.

Note that the communicated data is first copied to a separate buffer, and the send buffer is used again only during the next call to the routine. By that time the message from the former call has been delivered, and, as a result, there is no time delay in step 2 above. Step 5, however, may be associated with significant overhead, especially when there is an imbalance in the amount of floating-point operations performed on each processor. This issue may be partly mitigated by massive computations performed in step 4. Generally, step 4 contains the majority of floating-point operations, because the number of internal nodes in a mesh partition is much larger than that on a shared boundary.

Using the sorted data structure, all the vectors can be kept in a *communicated state*. Combining inter-processor communications with the matrix–vector product routine as detailed above produces values at the shared vector entries that are equal to the sum of the contributions of all the subdomains. This is identical to what a sequential algorithm would produce. As a result, to parallelize a sequential linear solver, only operations such as norm, inner product, and matrix–vector product need to be replaced by their parallel counterparts, without changing the main algorithm.

2.3 Implementation

Generally the ordering of unknowns is determined by the mesh generator rather than the LS itself. Hence, the linear system of equations that is received by the LS typically has a random order. In order to benefit from ordered data structure, data passed through the interface to the LS must be mapped. This operation has a negligible computational cost for vectors as it requires $O(n^i)$ operations on each processor.

Denoting $\boldsymbol{b}^i$ as the unsorted input and $\boldsymbol{x}^i_{\mathrm{s}}$ as the sorted output, the data mapping at the interface to the LS is as follows:

do $p = 1, \ldots, n^i$
  $b^i_{\mathrm{s}}(\mathcal{M}^i(p)) \leftarrow b^i(p)$

do $p = 1, \ldots, n^i$
  $x^i(p) \leftarrow x^i_{\mathrm{s}}(\mathcal{M}^i(p))$

All the computations inside the LS are now performed on the sorted data and no extra mappings are required.

Mapping the matrix $\boldsymbol{A}^i$ is more costly as the number of non-zeros, n^i_{nz}, is generally much larger than n^i. This extra computational cost may be avoided by adopting a specialized compressed sparse row (CSR) format [28]. Given an unsorted array of non-zero values on processor i, the matrix $\boldsymbol{A}^i$, an unsorted array of indices of the first nonzero element of each row, $\boldsymbol{I}^i$, and an unsorted array of the column indices of each entry of $\boldsymbol{A}^i$, $\boldsymbol{J}^i$, the sorted pointers are calculated as

do $p = 1, \ldots, n^i$
  $p_{\mathrm{s}} \leftarrow \mathcal{M}^i(p)$
  $I^i_{\mathrm{s}}(1, p_{\mathrm{s}}) \leftarrow I^i(p)$
  $I^i_{\mathrm{s}}(2, p_{\mathrm{s}}) \leftarrow I^i(p+1) - 1$

do $p = 1, \ldots, n^i_{\mathrm{nz}}$
  $J^i_{\mathrm{s}}(p) \leftarrow \mathcal{M}^i(J^i(p))$

Here $I^i_{\mathrm{s}}(1, p)$ and $I^i_{\mathrm{s}}(2, p)$ point to the first and last element of row p, and $J^i_{\mathrm{s}}(p)$ is the column index in the sorted format. With this transformation, there is no need to map $\boldsymbol{A}^i$, as the matrix-vector product $\boldsymbol{y}^i_{\mathrm{s}} = \boldsymbol{A}^i \boldsymbol{x}^i_{\mathrm{s}}$ may be expressed as:

$\boldsymbol{y}^i_{\mathrm{s}} \leftarrow 0$
do $p_{\mathrm{s}} = 1, \ldots, n^i$
  do $q = I^i_{\mathrm{s}}(1, p_{\mathrm{s}}), \ldots, I^i_{\mathrm{s}}(2, p_{\mathrm{s}})$
    $y^i_{\mathrm{s}}(p_{\mathrm{s}}) \leftarrow y^i_{\mathrm{s}}(p_{\mathrm{s}}) + A^i(q) x^i_{\mathrm{s}}(J^i_{\mathrm{s}}(q))$,

which is similar to a conventional sparse matrix-vector product.

Remarks

(1) The operator $\boldsymbol{I}^i_{\mathrm{s}}$ acts in the sorted domain and produces an unsorted codomain, while $\boldsymbol{J}^i_{\mathrm{s}}$ operates on the unsorted domain and produces a sorted codomain.
(2) Since $\mathcal{M}^i$ depends on the mesh connectivity and partitioning, $\mathcal{M}^i$, $\boldsymbol{I}^i_{\mathrm{s}}$, and $\boldsymbol{J}^i_{\mathrm{s}}$ are calculated only when the mesh or partitioning are changed. To solve multiple linear systems of equations that are based on the same mesh, as is the case of time dependent simulations, the LS is initialized only once by calculating $\mathcal{M}^i$, $\boldsymbol{I}^i_{\mathrm{s}}$, and $\boldsymbol{J}^i_{\mathrm{s}}$, and called each time with different $\boldsymbol{A}^i$ and $\boldsymbol{b}^i$. In this case, the extra cost associated with the LS initialization is typically negligible compared to the cost of the entire simulation.
(3) Only $\mathcal{M}^i$, $\boldsymbol{I}^i_{\mathrm{s}}$, and $\boldsymbol{J}^i_{\mathrm{s}}$ need to be stored, resulting in no additional significant memory requirements.
(4) In the case of multiple degrees-of-freedom per node, $\boldsymbol{A}^i$ and $\boldsymbol{b}^i$ may be allocated with an extra dimension that can be unrolled in operators for improved performance.

3 Results

The proposed algorithms are implemented in an efficient in-house linear solver written in Fortran that contains several iterative methods, such as the conjugate gradient (CG) [15] and generalized minimum residual (GMRES) [29] techniques, and a specialized algorithm for solving linear systems arising from the FEM discretization of the Navier–Stokes equations of incompressible flows [10,11]. The linear solver was used in several studies involving incompressible-flow and advection–diffusion equations in [8,12–14].

The results are verified for mathematical consistency by checking that all algebraic operations yield identical results for the parallel code and its serial counterpart. All the test cases were run on the Kraken machine at the University of Tennessee. Kraken uses 2.6 GHz AMD Opteron processors with 1.33 GB of memory per core. Compute nodes are interconnected via a Cray SeaStar2+ router. Kraken's default MPI library is employed in all computations, and a pure MPI programming paradigm is employed.

Three models are considered: small, medium, and large. The small model is a 3D cylinder that is meshed with 25k linear tetrahedral elements, producing 5.5k nodes and 81K non-zero entries in $\boldsymbol{A}$ (see Fig. 3a). The medium model is a 2D duct that is meshed with 14K bi-quadratic quadrilateral elements, producing 55k nodes and 890K non-zero entries in $\boldsymbol{A}$ (see Fig. 3b). The largest model is a 3D patient-specific aorta model from [31] that is meshed with 2.7M linear tetrahedral elements, producing 510K nodes and 7.4M non-zero entries in $\boldsymbol{A}$ (see Fig. 3c). The models are selected such that the number of unknowns varies by about one order of magnitude from case to case. The models are partitioned using ParMetis [18]. The number of partitions is chosen to be a power of two, that is, $n_{\mathrm{p}} = 2^i \; i \in \{0, \ldots, 11\}$.

To examine the influence of the compiler on the performance of the proposed method, we consider the small model, and use *pgi*, *Intel*, and *gnu* compilers to generate three separate executables. One hundred sparse matrix-vector products are performed for each case, and the results are presented in Fig. 4. The results show that all compilers give a similar performance when running in serial. However, as the number of

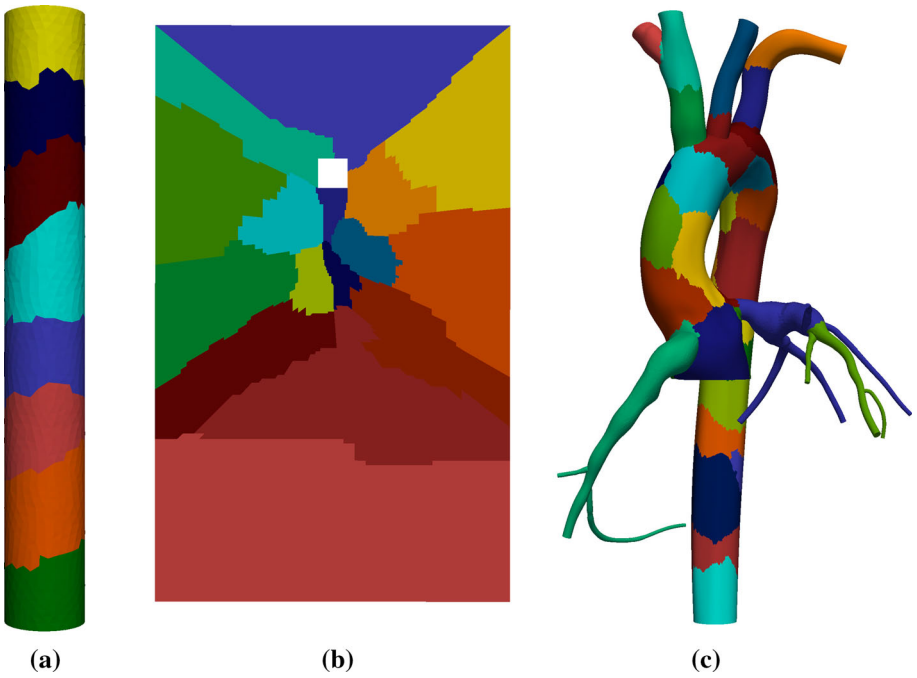

Fig. 3 Test cases: **a** 3D cylindrical model with $n_p = 8$ and $n = 5.5$k, **b** 2D model of a duct with $n_p = 16$ and $n = 55$k, and **c** 3D aortic model with $n_p = 32$ and $n = 510$k

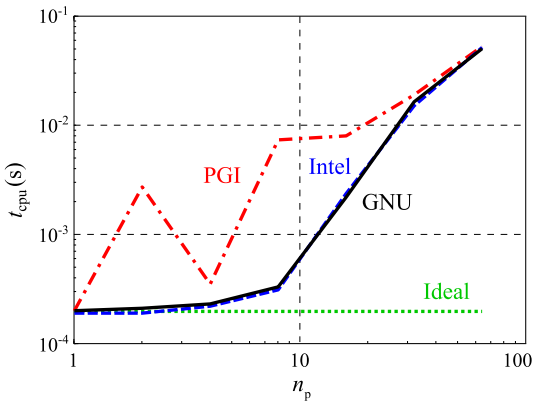

Fig. 4 Influence of the compiler on the performance of the sparse matrix-vector product. One hundred matrix-vector products were performed, and program execution time t_{cpu} is plotted versus the number of processors employed. Fortran compilers from *pgi*, *Intel*, and *gnu*, versions 11.9-0, 12.1.2, and 4.6.2, respectively, were used. These are the default compilers on Kraken at the present time. While *gnu* and *Intel* compilers gave very similar performance, the *pgi* compiler produced the least consistent results

processors is increased, the *pgi* compiler gives the least consistent performance. Based on this study, the *Intel* compiler is chosen for the rest of the computations presented in this paper.

Remark As a measure of the total computing effort, here we define t_{cpu}:

$$t_{cpu} = t_{elapsed} \times n_p, \tag{22}$$

where $t_{elapsed}$ is the time consumed by the n_p-processor system. In calculating the speedup, we use its classical definition, which is the time consumed by the sequential computation divided by the time consumed by the n_p-processor system. That is,

$$\text{Speedup} = \frac{t_{cpu}\,|_{n_p=1}}{t_{elapsed}}. \tag{23}$$

That can also be written as

$$\text{Speedup} = \frac{t_{cpu}\,|_{n_p=1}}{(t_{elapsed} \times n_p)/n_p}, \tag{24}$$

which translates to

$$\text{Speedup} = \frac{t_{cpu}\,|_{n_p=1}}{t_{cpu}} n_p. \tag{25}$$

To test the performance of the matrix-vector product operation, one hundred matrix-vector products were computed and t_{cpu} was measured for all three models. The results presented in Fig. 5 indicate that the maximum speedup depends on the size of the problem, increasing as the problem size increases. While at a lower number of processors the communication overhead is negligible and speedup is close to ideal, at a higher number of processors the communications overhead presents a major computational cost and leads to saturation. For the largest model the saturation was delayed to $n_p = 128$ in comparison with $n_p = 8$ for the smallest problem. Parallel

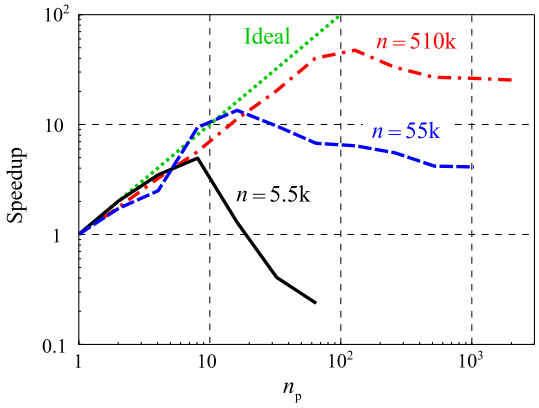

Fig. 5 Speedup in matrix vector product versus number of processors using matrices with $n = 5.5$k, $n = 55$k, $n = 510$k (Fig. 3)

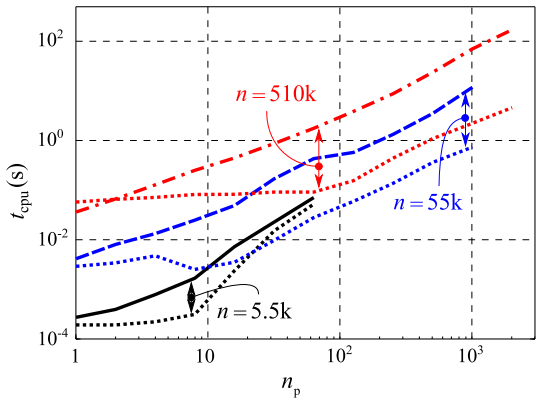

Fig. 6 Total cost of initialization (i.e., computing $\mathcal{M}^i$, $\boldsymbol{I}^i_s$, $\boldsymbol{J}^i_s$) and construction of the inter-processor communication data structures for the models shown in Fig. 3. *Solid*, *dashed*, and *dash-dotted lines* are initialization cost, and *dotted lines* are a single matrix–vector product cost, shown here for the reference. The *curves* are color-coordinated with the models employed

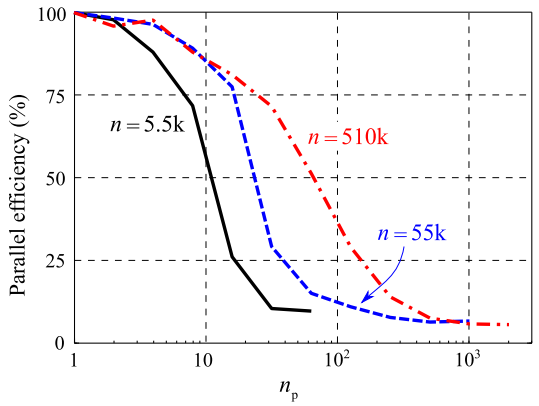

Fig. 7 Parallel efficiency of the matrix–vector product versus the number of partitions for the models shown in Fig. 3

scalability of the present method is compared with that of PETSc in the Appendix.

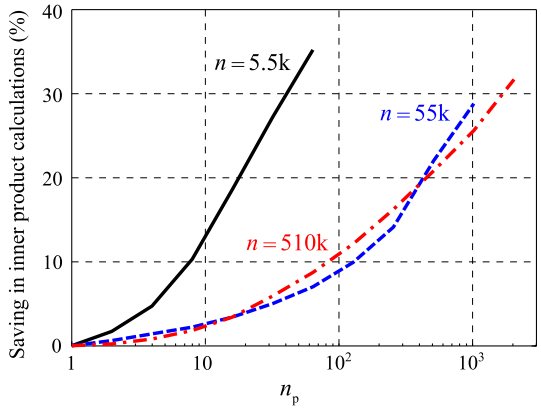

Fig. 8 Theoretical savings in norm and dot product calculation (i.e., $1 - \left(\sum_{i=1}^{n_p} n^i\right)^{-1} n$) versus the number of partitions for the models shown in Fig. 3

Table 1 Performance comparison of PETSc and the present method for the three models considered (see Fig. 5)

Case	n_p	n	t_{cpu} (ms)		Δ
			PETSc	Present	
a	8	5.5k	0.51	0.31	1.65
b	16	55k	9.7	3.5	3.77
c	64	510k	561	91	6.16

t_{cpu} is the total CPU time of computing a single matrix–vector product and Δ is the cost ratio between PETSc and the present method

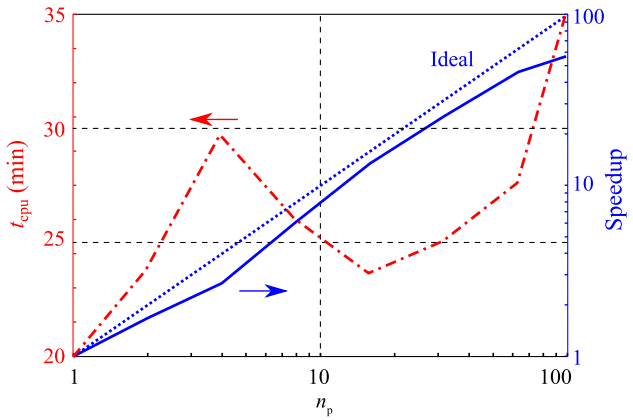

Fig. 9 The aorta model simulation cost and parallel speedup. *Red/dash-dot curve* (corresponding to the left y-axis) shows the cost, which is calculated as t_{cpu} consumed per time step. *Blue/solid curve* (corresponding to the right y-axis) is the parallel speedup. Ideal speedup is shown by a *blue/dotted* line

The initialization costs, that is, the costs of calculating $\mathcal{M}^i$, $\boldsymbol{I}^i_s$, $\boldsymbol{J}^i_s$, and constructing the data structures for inter-processor communications, are plotted in Fig. 6. The initialization costs may be up to 30 times higher than performing a single matrix-vector product. However, these costs increase almost linearly with n_p, resulting in wall-clock time that is

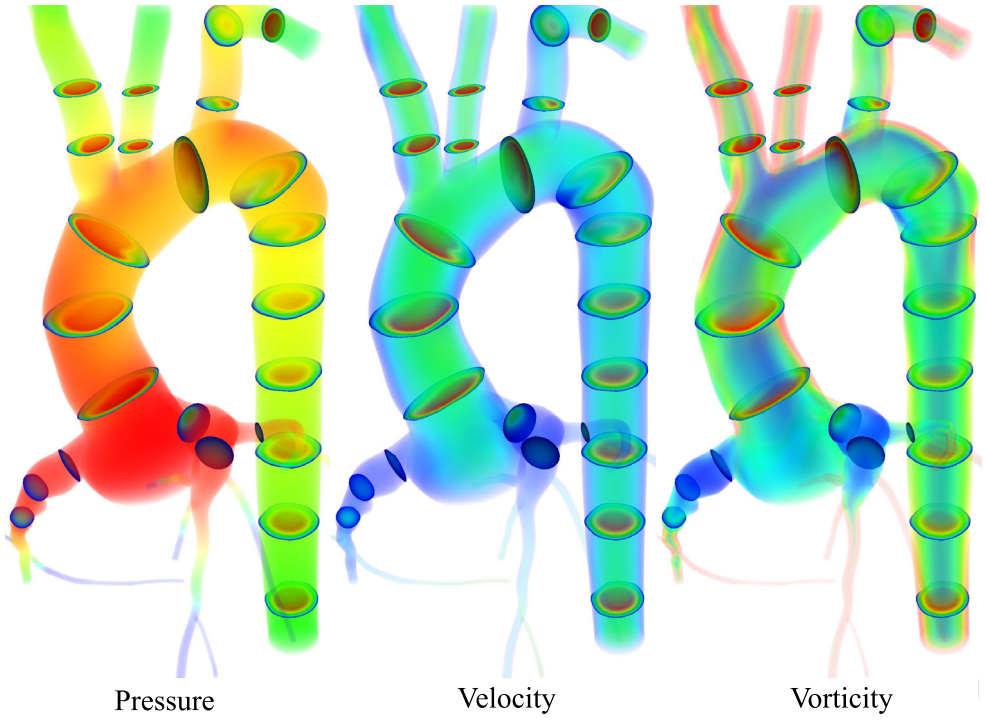

Fig. 10 Cross-sectional velocities with volume rendering of pressure (*left*), velocity magnitude (*center*), and vorticity magnitude (*right*) for the aorta model simulation

almost independent of the number of processors. For the three cases considered in this study, the initialization stage always took less than 0.1 s in wall-clock time, which is negligible, especially for time-dependent problems.

Parallel efficiency of the matrix–vector product, defined as the percentage ratio of the time spent on floating-point operations and t_{cpu} is calculated for the presented method (Fig. 7). As expected, there is a sharp drop in efficiency as n_{p} increases. The minimum total time for performing a matrix-vector product (i.e., the peak performance) generally occurs before parallel efficiency drops below 50 %.

The theoretical reduction in the number of floating point operations in a vector inner product is $n^i - n_{\text{o}}^i$ in partition i. The relative savings can be calculated as

$$\left(\sum_{i=1}^{n_{\text{p}}} n^i\right)^{-1} \sum_{i=1}^{n_{\text{p}}} (n^i - n_{\text{o}}^i), \tag{26}$$

and are plotted in Fig. 8. There are additional savings that are not included in Eq. (26), as it is not necessary to set n_{o}^i to n^i entries to zero in the new data structure. Increasing the number of partitions increases the number of nodes on the boundaries of partitions and leads to more significant savings. At peak performance (for n_{p} reported in Table 1), this translates to approximately a 10 % reduction in computations for all the test cases.

To test the performance of the proposed method in a CFD solver, blood flow in a patient-specific aorta model (see Fig. 3c) is simulated. A residual-based variational multiscale finite element formulation of incompressible flow is employed in conjunction with the Generalized-α time integration scheme [2,3,6,16,37,41]. Linear tetrahedral FEM is employed in the computation. The bi-partitioned iterative method is employed for solving the coupled velocity-pressure linear system [11]. Physiologically realistic resistance boundary conditions are imposed at all outlets, and steady inflow boundary condition is imposed at the inlet. For the resistance boundary conditions, a specialized preconditioner is employed to improve the convergence rate of the iterative solver [10]. The time step size is set to 1 ms and the simulation is continued for 10 time steps. The simulation cost per time step along with parallel speedup is shown in Fig. 9. The figure shows good scalability of the present method for up to 64 processors, where $n^i \approx 8$k. In terms of *elapsed real time* (not t_{cpu}), one time step takes 21 sec at $n_{\text{p}} = 100$ in comparison to 20 min at $n_{\text{p}} = 1$ (Fig. 10).

4 Conclusions

A method for iterative linear equation solvers is presented that allows a systematic conversion of sequential algebraic operations, such as vector inner product and matrix-vector product, to their parallel counterparts. Improved parallel per-

formance is obtained by introducing a mapping that separates the shared entries from the unshared entries. This allows one to combine non-blocking communications with computations in matrix-vector products, and to avoid performing repeated computations on the shared entries in vector inner products. The initialization and use of the new data structures have minimal time and memory requirements. The proposed method was tested on a varying number of processors using three computational models of different size and complexity, showing very good performance. Testing the proposed method on a patient-specific blood flow problem with $n = 510$k showed good scalability up to 64 cores, with as few as 8,000 nodes per core.

Acknowledgments Funding for this work was provided by a Leducq Foundation Network of Excellent Grant, a Burroughs Welcome Fund Career Award at the Scientific Interface, and the NIH grant RHL102596A. The second author was supported by the NSF CAREER award OCI-105509. The computational resources were provided by the national XSEDE program.

Appendix

The performance of the present method is compared with that of PETSc, a widely adopted linear algebra library of routines [1]. A Cray PETSc 3.2.00 release, equivalent to a 3.2-p5 release by the Argonne National Laboratory, is used for comparison. Since there may be significant differences in the implementation of iterative linear algebra algorithms, only matrix-vector product operation is considered in this comparison study. Our choices for the PETSc matrix type, its initialization and assembly, and also the matrix-vector product operation are included here for completeness:

```
Mat A
Vec x, b
VecCreate(PETSC_COMM_WORLD,&x)
VecSetSizes(x, n^i, n)
VecSetFromOptions(x)
VecDuplicate(x,&b)
VecSetValues(x, n^i, a^i, x̃,INSERT_VALUES)
VecAssemblyBegin(x)
VecAssemblyEnd(x)
MatCreate(PETSC_COMM_WORLD,&A)
MatSetSizes(A, n^i, n^i, n, n)
MatSetFromOptions(A)
MatSetUp(A)
for p = 1, ..., n^i
   for q = I^i(p), ..., I^i(p + 1) − 1
      t(q − I^i(p)) = a^i(J^i(q))
   MatSetValues(A, 1, a^i(p), I^i(p+1)−I^i(p), t, Ã,INSERT_VALUES)
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY)
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY)
// Start time recorder
for 1, ..., 100
   MatMult(A,x,b)
// Stop time recorder
```

In the above pseudo-code $\tilde{\boldsymbol{x}}$ and $\tilde{\boldsymbol{A}}$ are the memory locations of the vector $\boldsymbol{x}$ and matrix $\boldsymbol{A}$, $\boldsymbol{t}$ is a temporary array, and $\boldsymbol{I}^i$ and $\boldsymbol{J}^i$ are the C-compatible integer arrays with 0 as the first entry.

Computations associated with Fig. 5 are repeated using PETSc. One hundred matrix-vector products are computed and t_{cpu} is measured. To compare the methods in terms of minimal time to completion of a matrix-vector product operation, the cases of $n_{\text{p}} = 8$, 16, and 64 are considered for the small, medium, and large model, respectively (see Table 1). These correspond to near-peak performance of both techniques (see Fig. 5). The results show that by increasing the problem size the difference between the peak performance of the present method and PETSc becomes more apparent: The higher the number of partitions, the better the present method performs relative to PETSc. The improvement for the small, medium, and large model is 65, 177, and 516 %, respectively. Repeating the computations on a different machine and using a different version of PETSc had minimal effect on the results. Note that, among the several options available, we used a basic PETSc matrix type in this comparison. PETSc results may depend on the matrix type, however, this was not investigated here.

References

1. Balay S, Brown J, Buschelman K, Eijkhout V, Gropp W, Kaushik D, Knepley M, Curfman McInnes L, Smith B, Zhang H (2013) PETSc Users Manual Revision 3.4, 2013
2. Bazilevs Y, Calo VM, Cottrell JA, Hughes TJR, Reali A, Scovazzi G (2007) Variational multiscale residual-based turbulence modeling for large eddy simulation of incompressible flows. Comput Methods Appl Mech Eng 197(1–4):173–201
3. Bazilevs Y, Takizawa K, Tezduyar TE (2013) Computational fluid-structure interaction: methods and applications. Wiley, New York
4. Behr M, Johnson A, Kennedy J, Mittal S, Tezduyar T (1993) Computation of incompressible flows with implicit finite element implementations on the Connection Machine. Comput Methods Appl Mech Eng 108:99–118
5. Behr M, Tezduyar TE (1994) Finite element solution strategies for large-scale flow simulations. Comput Methods Appl Mech Eng 112:3–24
6. Brooks AN, Hughes TJR (1982) Streamline upwind/Petrov–Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations. Comput Methods Appl Mech Eng 32(1–3):199–259
7. Elman H, Silvester D, Wathen A (2014) Finite elements and fast iterative solvers with applications in incompressible fluid dynamics. Oxford University Press, New York
8. Esmaily-Moghadam M, Bazilevs Y, Hsia TY, Vignon-Clementel I, Marsden AL (2011) A comparison of outlet boundary treatments for prevention of backflow divergence with relevance to blood flow simulations. Comput Mech 48(3):277–291
9. Esmaily-Moghadam M, Bazilevs Y, Marsden AL (2013) Low entropy data mapping for sparse iterative linear solvers. In Proceedings of the conference on extreme science and engineering discovery environment: gateway to discovery, p 2. ACM, 2013

10. Esmaily-Moghadam M, Bazilevs Y, Marsden AL (2013) A new preconditioning technique for implicitly coupled multidomain simulations with applications to hemodynamics. Comput Mech 52:1141–1152. doi:10.1007/s00466-013-0868-1
11. Esmaily-Moghadam M, Bazilevs Y, Marsden AL (2014) A bi-partitioned iterative algorithm for solving linear systems arising from incompressible flow problems. Comput Methods Appl Mech Eng, in review
12. Esmaily-Moghadam M, Hsia T-Y, Marsden AL (2013) A non-discrete method for computation of residence time in fluid mechanics simulations. Phys Fluids. doi:10.1063/1.4819142
13. Esmaily-Moghadam M, Migliavacca F, Vignon-Clementel IE, Hsia TY, Marsden AL (2012) Optimization of shunt placement for the Norwood surgery using multi-domain modeling. J Biomech Eng 134(5):051002
14. Esmaily-Moghadam M, Vignon-Clementel IE, Figliola R, Marsden AL (2013) A modular numerical method for implicit 0D/3D coupling in cardiovascular finite element simulations. J Comput Phys 224:63–79
15. Hestenes MR, Stiefel E (1952) Methods of conjugate gradients for solving linear systems. J Res Natl Bur Stand 49:409–436
16. Jansen KE, Whiting CH, Hulbert GM (2000) A generalized-[alpha] method for integrating the filtered Navier-Stokes equations with a stabilized finite element method. Comput Methods Appl Mech Eng 190(3—-4):305–319
17. Johnson AA, Tezduyar TE (1997) 3D simulation of fluid-particle interactions with the number of particles reaching 100. Comput Methods Appl Mech Eng 145:301–321
18. Karypis G, Kumar V (2009) MeTis: unstructured graph partitioning and sparse matrix ordering system, Version 4.0. http://www.cs.umn.edu/~metis
19. Kennedy JG, Behr M, Kalro V, Tezduyar TE (1994) Implementation of implicit finite element methods for incompressible flows on the CM-5. Comput Methods Appl Mech Eng 119:95–111
20. Kuck DJ, Davidson ES, Lawrie DH, Sameh AH (1986) Parallel supercomputing today and the cedar approach. Science 231(4741):967–974
21. Manguoglu M, Sameh AH, Saied F, Tezduyar TE, Sathe S (2009) Preconditioning techniques for nonsymmetric linear systems in the computation of incompressible flows. J Appl Mech 76(2):021204
22. Manguoglu M, Sameh AH, Tezduyar TE, Sathe S (2008) A nested iterative scheme for computation of incompressible flows in long domains. Comput Mech 43(1):73–80
23. Manguoglu M, Takizawa K, Sameh AH, Tezduyar TE (2010) Solution of linear systems in arterial fluid mechanics computations with boundary layer mesh refinement. Comput Mech 46(1):83–89
24. Manguoglu M, Takizawa K, Sameh AH, Tezduyar TE (2011) Nested and parallel sparse algorithms for arterial fluid mechanics computations with boundary layer mesh refinement. Int J Numer Methods Fluids 65(1–3):135–149
25. Manguoglu M, Takizawa K, Sameh AH, Tezduyar TE (2011) A parallel sparse algorithm targeting arterial fluid mechanics computations. Comput Mech 48(3):377–384
26. Nigro N, Storti M, Idelsohn S, Tezduyar T (1998) Physics based GMRES preconditioner for compressible and incompressible Navier-Stokes equations. Comput Methods Appl Mech Eng 154:203–228
27. Polizzi E, Sameh AH (2006) A parallel hybrid banded system solver: the SPIKE algorithm. Parallel Comput 32(2):177–194
28. Saad Y (2003) Iterative methods for sparse linear systems. In: SIAM, 2003
29. Saad Y, Schultz MH (1983) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. Technical Report YALEU/DCS/RR-254, Department of Computer Science, Yale University, Yale
30. Sameh AH, Kuck DJ (1978) On stable parallel linear system solvers. J ACM 25(1):81–91
31. Sengupta D, Kahn A, Burns J, Sankaran S, Shadden S, Marsden A (2012) Image-based modeling of hemodynamics in coronary artery aneurysms caused by Kawasaki disease. Biomech Model Mechanobiol 11:915–932
32. Tezduyar T, Aliabadi S, Behr M, Johnson A, Mittal S (1993) Parallel finite-element computation of 3D flows. Computer 26(10):27–36
33. Tezduyar TE (2001) Finite element methods for flow problems with moving boundaries and interfaces. Arch Comput Methods Eng 8:83–130
34. Tezduyar TE (2007) Finite elements in fluids: special methods and enhanced solution techniques. Comput Fluids 36:207–223
35. Tezduyar TE, Behr M, Aliabadi SK, Mittal S, Ray SE (1992) A new mixed preconditioning method for finite element computations. Comput Methods Appl Mech Eng 99:27–42
36. Tezduyar TE, Liou J (1989) Grouped element-by-element iteration schemes for incompressible flow computations. Comput Phys Commun 53:441–453
37. Tezduyar TE, Mittal S, Ray SE, Shih R (1992) Incompressible flow computations with stabilized bilinear and linear equal-order-interpolation velocity-pressure elements. Comput Methods Appl Mech Eng 95:221–242
38. Tezduyar TE, Sathe S (2004) Enhanced-approximation linear solution technique (EALST). Comput Methods Appl Mech Eng 193:2033–2049
39. Tezduyar TE, Sathe S (2005) Enhanced-discretization successive update method (EDSUM). Int J Numer Methods Fluids 47:633–654
40. Tezduyar TE, Sathe S (2007) Modeling of fluid-structure interactions with the space-time finite elements: Solution techniques. Int J Numer Methods Fluids 54:855–900
41. Tezduyar TE (2003) Computation of moving boundaries and interfaces and stabilization parameters. Int J Numer Methods Fluids 43(5):555–575
42. Tezduyar TE, Sameh AH (2006) Parallel finite element computations in fluid mechanics. Comput Methods Appl Mech Eng 195(13):1872–1884
43. Washio T, Hisada T, Watanabe H, Tezduyar TE (2005) A robust preconditioner for fluid-structure interaction problems. Comput Methods Appl Mech Eng 194:4027–4047