Parallel performance of the coarse space linear vertex solver and low energy basis preconditioner for spectral/hp elements


ABSTRACT

The big bottleneck in scaling PDE-based codes to petaflop computing is scalability of effective preconditioners. We have developed and implemented an effective and scalable low energy basis preconditioner (LEBP) for elliptic solvers, leading to computational savings of an order of magnitude with respect to other preconditioners. The efficiency of LEBP relies on the implementation of parallel matrix-vector multiplication required by coarse solver to handle the h-scaling. We provide details on optimization, parallel performance and implementation of the coarse grain solver and show scalability of LEBP on the IBM Blue Gene and the Cray XT3.

1. Introduction

Scalability of highly effective preconditioners on thousands of processors is an open problem that had no satisfactory solution for many matrix problem arising from the discretisation of partial differential equations. As the effectiveness of the preconditioner increases so does the global interdependencies reflecting in a sense the multi-scale nature of the problem and hence the poor scaling. To the best of our knowledge, there are currently no effective preconditioners for the new generation of modal spectral/hp element discretisations that scale well on more than one thousand processors.

Several effective preconditioners for spectral element Navier–Stokes solvers can be found in the literature, but very few of these preconditioners are also scalable. Tufo and Fischer [1] presented a scalable parallel solver for the incompressible Navier–Stokes equation, for the first (nodal) generation of spectral elements. Performance of the solver was evaluated on the Intel ASCI-Red, CRAY T3E-600 and SGI ASCI-Blue computers with 333 MHz processors. Good scalability was observed for simulations with relatively high order polynomial approximation. Bergen et al. [2] used hierarchical hybrid grid (HHG) to solve efficiently large scale linear finite element problem. The HHG approach is essentially a geometric multigrid method. Good scalability and also good solution time was achieved for solution of a large problem on up to 1024 SGI AtlixTM 3700 CPUs (1.6 GHz Itanium 2 processors). Lottes and Fischer [3] studied performance of the variations of multigrid method applied to spectral element nodal discretizations of the Helmholtz equation. Several methods considered in their work resulted in convergence rates comparable to regular (Cartesian) grid based multigrid methods. Although effective for non-deformed spectral elements, geometric multigrid is essentially a sequential approach and can not lead to a high parallel efficiency on petaflop
computers. Pavarino et al. [4] developed overlapping Schwarz methods for nodal triangular and quadrilateral spectral elements. Their results show that it is possible to obtain convergence rate independent of the order of polynomial expansion and number of elements.

A low energy basis preconditioner (LEBP) for elliptic substructured solvers was proposed by Bica [5] and later implemented by Sherwin and Casarin [6] for modal spectral/hp elements. In this work we present a parallel implementation of the LEBP appropriate for a large number of processors and investigate its performance. Specifically, we discuss in detail implementation of a parallel coarse space linear vertex solver. Our results show that LEBP is very effective in simulations on more than one thousand processors while it exhibits similar scalability to a diagonal preconditioner. The goal of this project was to develop a scalable and efficient coarse space linear vertex solver required by a LEBP for elliptic substructured solver. Guided by the information obtained with profiling tools on IBM Blue Gene/L and DataStar (Power4+/Federation) computer at SDSC we were able to make optimizations by redesigning the communication in several bottleneck routines. Profiling code simultaneously on several computational platforms allowed us to perform differential optimization. As a result, the code implemented with low energy basis preconditioner and the optimized coarse space linear vertex solver now scales well on thousands of processors.

The spectral/hp element code NEKTAR is employed in our studies [7]. The computational domain consists of structured or unstructured grids or a combination of both, similar to those employed in standard finite element and finite volume methods. In each element the solution is approximated in terms of hierarchical mixed order Jacobi polynomial expansions. This provides a way of hierarchically refining the numerical solution by increasing the order of the expansion (p-refinement) within every element without the need to regenerate the mesh, thus avoiding a potential significant overhead cost associated with remeshing. Domain partitioning, required by the parallel solver, is done by Metis [10]. NEKTAR employs up to third-order accurate semi-implicit time integration scheme. A high-order splitting scheme [8] is adopted, that decouples the velocity and pressure fields requiring only the inversion of three Helmholtz operators for the velocity components (in three-dimensions) and one Poisson operator for the pressure; for details see Appendix A. Due to the matrix sparsity arising from symmetric linear operators, iterative solutions based on preconditioned conjugate gradient (PCG) are typically preferred. The effectiveness of preconditioner is normally estimated by reduction in iteration count. However, the parallel efficiency of a given preconditioner is also strongly affected by the additional computational cost associated with the preconditioning step and by the volume of communication involved. In this paper we measure the effectiveness of preconditioner by monitoring the reduction in the average cpu-time required for one time step in large 3D simulations of flow in arterial geometries.

The following computational platforms were used for the benchmarking and code developing:

- **IBM Blue Gene of San Diego Supercomputing Center (SDSC):** This computer is housed in three racks with 3072 compute nodes. Each node has two PowerPC processors that run at 700 MHz and share 512 MB of memory. All compute nodes are connected by two high-speed networks: a 3D torus for point-to-point message passing and a global tree for collective message passing.
- **IBM Power4+/Federation of SDSC:** This computer has eight-way P655+ 1.5 GHz and 32-way P690 1.7 GHz compute nodes with 32 and 128 GB of memory, respectively. In our study we used the P655 compute nodes.
- **Cray XT3 MPP system of Pittsburgh Supercomputing Center (PSC) with 2068 compute nodes linked by a custom-designed interconnect. Each compute node has two 2.6 GHz AMD Opteron processors with its own cache, and shared 2 GB of memory and the network connection. The nodes are connected in a three-dimensional torus using a HyperTransport link to a dedicated Cray SeaStar communications engine.**
- **Cray XT3 MPP of US Army Engineer Research and Development Center (ERDC):** The computer has 4160 nodes, each containing one 2.6-GHz AMD Opteron 64-bit dual-core processor and 4 GB of shared memory. The nodes are connected in a three-dimensional torus using a HyperTransport link to a dedicated Cray SeaStar communications engine.

The paper is organized as follows: in Section 2 we first review the hierarchical basis implemented in NEKTAR. In Section 3 we overview the low energy preconditioner, extend the formulation of [6], and introduce an improvement in performance of the LEBP for hybrid meshes with prismatic elements. In Section 4 we provide details on the construction and implementation of parallel coarse space solver – the primary bottleneck in the scaling of the low energy preconditioner. In Section 5 we present results on parallel efficiency of the LEBP. In Section 6 we conclude with a brief summary. In Appendices A–C we provide additional information on the time-stepping scheme employed in NEKTAR, construction of LEBP and communication patterns implemented for parallel matrix-vector product in NEKTAR.

## 2. Expansion basis in NEKTAR

The computational domain in NEKTAR consists of tetrahedra, hexahedra, prisms, pyramids or a combination of these. Within each element the solution is approximated in terms of hierarchical, mixed order, semi-orthogonal Jacobi polynomial expansions [7]. It is hierarchical in a sense that the modes are separated into vertex (linear term), edge, face and bubble (high-order terms) modes. In Fig. 1 we provide an illustration of the domain decomposition and polynomial basis employed in NEKTAR.
The polynomial expansion basis within each element is decomposed into interior and boundary modes (vertex, edge and face) to help construct a global $C^0$-continuity. The interior modes have zero support on the elemental boundaries, thus the boundary and interior degrees of freedom can be numerically decoupled through a technique known as substructuring where the Schur complement of the boundary system is constructed. The boundary degrees of freedom, corresponding to adjacent boundary and interior degrees of freedom can be numerically decoupled through a technique known as substructuring when pieced together under the mapping $\mathcal{P}$.

The polynomial expansion basis within each element is decomposed into interior and boundary modes (vertex, edge and face) to help construct a global $C^0$-continuity. The interior modes have zero support on the elemental boundaries, thus the boundary and interior degrees of freedom can be numerically decoupled through a technique known as substructuring where the Schur complement of the boundary system is constructed. The boundary degrees of freedom, corresponding to adjacent elements, are coupled due the requirement of $C^0$-continuity. Moreover, there is a strong coupling between the vertex, edge and face modes within each element. It is this strong coupling between the boundary degrees of freedom that leads to solution of large linear system with high condition number.

The dependence of the condition number of the preconditioned Schur complement of the Laplacian matrix on the order of polynomial approximation was reported in [6]. The condition number of 3D Laplacian matrix scales with the order of polynomial approximation was reported in [6]. The condition number of 3D Laplacian matrix scales with the order of polynomial expansion as $\kappa \propto P^{3/2}$ for tetrahedral and prismatic meshes with fixed number of elements reciprocally, whereas the condition number of the preconditioned (with LEBP) Schur complement scales as $\kappa \propto (1 + \log(P))^2$. Bica [5] and Pavarino and Widlund [9] presented a theoretical proof and numerical verification for that polylogarithmic growth of $\kappa$ for solution of 3D problems with the $p$-version finite element method based on continuous, piecewise polynomial expansions.

As we will see in the next section, the idea of the LEBP is to weaken the coupling of the boundary modes, enhance the diagonal dominance of the matrix system and then to apply a block-preconditioning technique for the vertex, edge and face degrees of freedom.

3. Low energy basis preconditioner (LEBP)

3.1. Formulation

Consider the elliptic boundary value problem

$$\nabla^2 u(x) + \lambda u(x) = f(x); \quad \lambda \leq 0, \quad (1)$$

defined on computational domain $\Omega$ which is discretized into $N_d$ non-overlapping spectral elements; the computational subdomain, associated with a particular element, is denoted by $\Omega_d$. A standard spectral/$hp$ element [7] is defined on a local to this element, system of coordinates $\xi$, which can be mapped to the global coordinate system $x = \mathbf{x}_i(\xi)$. Then, a standard spectral/$hp$ element [7] spatial approximation of $u$ is given by

$$u^\delta(x) = \sum_{i=1}^{N_d} \hat{u}_i \phi_i(x) = \sum_{e=1}^{N_d} \sum_{j=1}^{\dim(V_e)} \hat{u}^e_j \phi^e_j(\mathbf{x}_e(\xi)). \quad (2)$$

where $N_d$ is a total number of degrees of freedom and $\phi_i(\mathbf{x}_i(\xi))$ are polynomials defined in a space $V^e$ of order $P$, which when pieced together under the mapping $\mathbf{x}_i(\xi)$ make a $C^0$ continuous (global) expansion $\Phi(\mathbf{x})$. The superscript $\delta$ emphasize that we use a finite (truncated) space.

Using Galerkin discretization of (1), i.e., find $u^\delta \in V^\delta$ such that

$$\mathcal{L}(v, u) = \int_{\Omega} \nabla v^\delta \cdot \nabla u^\delta + \lambda v^\delta u^\delta d(x) = \int_{\Omega} v^\delta f d(x) \quad \forall v^\delta \in V^\delta \quad (3)$$

we obtain the weak formulation of (1). Following the standard Galerkin formulation adopted in finite element methods and letting $v^\delta = \Phi_i (i = 1, \ldots, \dim(V^e))$, problem (1) can be recast into a matrix equation

$$\mathbf{H}u = f,$$

where $\mathbf{H}(i,j) = \int_{\Omega} \nabla \phi_i(x) \nabla \phi_j(x) + \lambda \phi_i(x) \phi_j(x) d(x)$ denotes the Helmholtz operator and $f(i) = \int_{\Omega} \Phi_i(x)f d(x)$. 

![Fig. 1. Illustration of the unstructured surface grid and the polynomial basis employed in NEKTAR. The solution domain is decomposed into non-overlapping elements. Within each element the solution is approximated by vertex, edge, face and (in 3D) interior modes. The shape functions associated with the vertex, edge and face modes for fourth-order polynomial expansion defined on triangular and quadrilateral elements are shown in color.](image-url)
In terms of implementation, the global matrix $H$ is typically assembled from elemental contributions $H^i$ where

$$H^i(i,j) = \int_{\Omega^i} \nabla \phi_i^e(x) \nabla \phi_j^e(x) + \lambda \phi_i^e(x) \phi_j^e(x) \ d(x)$$

and $\phi_i^e(x)$ denote the elemental representations of the elemental expansion which when assembled globally make $\Phi_i(x) [7]$. The elemental approximation, $u_i^e$, can be expressed by different sets of polynomial expansion from the same space $(V^e)$

$$u_i^e(x(\xi)) = \sum_{i=1}^{\dim(V^e)} u_{i1} \phi_i^e(\xi) \equiv \sum_{i=1}^{\dim(V^e)} u_{i2} \phi_i^e(\xi).$$

and therefore it is possible to define a matrix transform $C$ from basis $\Phi_i$ to $\Phi_i^e$, i.e.,

$$\phi_i^e = C \phi_i.$$

and this process is possible even if a close form expression for $\phi_i^e$ is not available. Due to the linearity of the operation, $C$ can be applied to transform the operator $H^i$ computed with different expansion bases, i.e.,

$$H_i^e = C H_i C^T, H_i = C^{-1} H_i^T (C^T)^{-1}.$$

Analogous to the elemental decomposition of $H$ into components $H^i$, we introduce elemental contributions to $\hat{u}$ and $f$ denoted as $\hat{u}^i$ and $f^i$. Further decomposing $\hat{u}^i$ and $f^i$ into contributions associated with the boundary ($\hat{u}_b^i$) and interior ($\hat{u}_i^i$) modes we obtain:

$$\begin{bmatrix} H_{bb} & H_{bi} \\ H_{ib} & H_{ii} \end{bmatrix} \begin{bmatrix} \hat{u}_b^i \\ \hat{u}_i^i \end{bmatrix} = \begin{bmatrix} f_b^i \\ f_i^i \end{bmatrix}.$$

Since the support of the interior modes, which is used to construct $H_i^e$ are non-overlapping, their contribution to $H$ are decoupled from each other. It is therefore an excepted practice to construct the Schur complement $S = H_{bb} - H_{bi} H_{ii}^{-1} H_{ib}$ and solve directly for the boundary degrees of freedom, which when known can be used to recover the interior degrees of freedom.

We therefore restrict our discussion to preconditioning only the Schur complement system and define a transformation matrix $C$ as

$$C = \begin{bmatrix} R & 0 \\ 0 & I \end{bmatrix}$$

then, after substructuring the matrix system $H^e = C H^i C^T$, we obtain that the Schur complement of $H^e$ is related to the Schur complement of $H^i$ by

$$S^e = R S^i R^T,$$

where $S^e, S^i$ are the Schur complements of $H^e$ and $H^i$, respectively. Our task is now to define an appropriate transformation matrix $R$ in order to obtain $S^e$ with a predominantly diagonal structure. Details on the structure and numerical construction of the matrix $R$ can be found in Appendix B and in [6].

We note that in order to reduce significantly the number of iterations, we want to design a preconditioner whose eigen-spectrum is close to the one of the original operator. However, the efficiency of a parallel solver is also strongly affected by the dominance of the modified Schur complement is essential.

The LEBP is developed by numerically constructing a “low energy” basis (i.e., $\int \phi_i^2 \phi_j^2 \ d\Omega < \int \phi_i^2 \phi_j^2 \ d\Omega$, particularly for $i \neq j$) related to the transformation matrix $R$. This is achieved by considering the elemental matrices arising for a given spectral/hp problem within a standardized region and numerically constructing a new low energy basis, where coupling between each vertex mode with respect to the edge and face modes, and also the edge modes with respect to the face modes is minimized. Due to the influence of local elemental mappings $X(\xi)$ the transformed global matrix $H$ will not maintain the orthogonality introduced by the low energy basis on $H^e$. However, the transformed system is now diagonally dominant and so block diagonal preconditioning of this transformed matrix leads to a polylogarithmic scaling of the condition number with respect to polynomial order $P$ for reasonably well behaved matrices. Since the number of iterations of the conjugate gradient methods scales as a square root of the condition number this approach leads to an effective $p$-type preconditioner. Further, the local communication associated with edges and faces means that this preconditioner is very suitable for parallelization.

An example of numerically-derived basis for a low energy vertex mode is shown in Fig. 2 [6]. The standard linear finite element vertex mode is shown in Fig. 2a and the numerically constructed low-energy vertex mode is shown in Fig. 2b. As depicted in Fig. 3 the transformed Schur complement system $S^e$ has much more diagonal-dominant structure than the original $S^i$.

When constructing the global Schur complement $S$ from $S^e$, we now choose to design a preconditioner which inverts blocks corresponding to degrees of freedom along each global edge and face since these are relatively small and easily inverted.
The h-scaling is most effectively handled by directly inverting the coarse linear finite element space block associated with the spectral/hp discretization; here we define the coarse linear finite element space as a space of the vertex degrees of freedom. The definition of the coarse linear finite element space block is given in 4.1. Thus, the additive Schwartz preconditioner is a combination of the coarse space linear vertex block and the block-diagonal LEBP, as described in Section 2.6 of [6]. Since in these types of methods we typically have quite coarse grids compared to classical finite element discretization, a direct inversion is still tractable from a memory standpoint (we should note that methods for direct solution of the equations on a linear finite element mesh may be also based on approaches we describe in this paper). However, extra algorithmic work is required to make this coarse space linear vertex solve scale from a parallelization/communication point of view. In Section 4 we discuss the implementation of parallel direct solver associated with the coarse grid preconditioning.

3.2. Low energy basis preconditioner for prismatic elements

For many applications, e.g. in boundary layers, we use a single layer of prismatic elements to capture the boundary layer of the flow. The interior space is then filled with tetrahedrons. In this section we present an improvement in constructing LEBP for prismatic elements used in this type of meshing strategy. In the original work of [6] the construction of LEBP was based on considering a standard prismatic element, which has two equilateral triangular surfaces connected by three quadratic faces which all have edges of a similar length. The equilateral shape of the triangular face was to ensure the global continuity of the low energy shape functions between the prismatic and tetrahedral elements.

In the following, we modify the aspect ratio of the standard prismatic element by varying the distance between the triangular faces (thickness of the prismatic element). The reshaping of element results in a better fit of the standard element to that used in the physical mesh. We denote by $\alpha$ the parameter by which we scale the distance between triangular faces of the standard prismatic element. In general, not all elements in the original mesh have the same aspect ratio, which makes adjustment of $\alpha$ not trivial.

To check the effect of $\alpha$ on the convergence rate we performed a steady flow simulation in the domain shown in Fig. 1. The mesh is constructed from one layer of prismatic elements while the rest are tetrahedral elements. The dimensions of the prismatic element are summarized in Table 1. In Fig. 4 we present the number of iterations required by the Poisson solver...
for the pressure and Helmholtz solver for the streamwise ($w$) velocity component. In Fig. 5 we show the number of iterations required by the linear solvers with respect to polynomial order and parameter $\alpha$. In a case of elemental Mass matrix, the factor $\alpha$ scales the Jacobian of the standard element only, which is equivalent to scaling the Mass matrix with a constant. This scaling does not modify the ratio of the largest to smallest eigenvalues $\lambda_{\text{MAX}}/\lambda_{\text{MIN}}$. If all elements in the mesh have the same Jacobian and the same scaling factor $\alpha$, then the condition number of the global, statically condensed Mass matrix, will not be affected. In the case of Stiffness matrix, the parameter $\alpha$ appears not only in the Jacobian but its inverse appears also in the derivatives taken along the AD direction (see Fig. 1), thus it affects the $\lambda_{\text{MAX}}/\lambda_{\text{MIN}}$ ratio. In a hybrid mesh, where the Jacobian of all prismatic elements are scaled by $\alpha$ while the Jacobian of the tetrahedral elements remains unmodified, the condition number of the global Mass and Stiffness matrices changes. The Helmholtz operator is a weighted linear combination of the Mass and the Stiffness operators, hence its condition number will depend on both matrices.

Table 1

<table>
<thead>
<tr>
<th>Edge</th>
<th>AB</th>
<th>AC</th>
<th>AD</th>
<th>DE</th>
<th>DF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (original)</td>
<td>0.717</td>
<td>0.694</td>
<td>0.133</td>
<td>0.717</td>
<td>0.738</td>
</tr>
<tr>
<td>Length (standard)</td>
<td>1</td>
<td>1</td>
<td>$\alpha$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 4. Hybrid mesh: performance of LEBP versus time step as a function of parameter $\alpha$ for Poisson (upper) and Helmholtz (lower) solvers. Simulation of a steady flow in a domain presented in Fig. 1.

Fig. 5. Hybrid mesh: performance of LEBP as a function of parameter $\alpha$. Simulation of a steady flow in a domain presented in Fig. 1. Mean number of iterations required by the last ten time-steps of Fig. 1.
As we observe in Fig. 5 the number of iterations required to solve the Helmholtz equations for the three velocity components is not very sensitive to \( \lambda \), while the number of iterations required by the Poisson solver strongly depends on \( \lambda \). For these types of meshes, the thickness of the standard prismatic element leads to a better approximation of the shape of the original elements and, as a result, to a lower iteration count, ultimately leading to computational savings.

4. Parallel coarse space linear vertex solver

In this section we first discuss the partitioning of global linear vertex degrees of freedom and formulation of the Schur complement for the coarse vertex solve. Second, we compare several numerical approaches to tackle the solution of a linear system required by the coarse vertex solve. Third, we provide details on the parallel construction of the Schur complement for the coarse vertex solve during the preprocessing stage. Finally, we discuss different algorithms for parallel matrix–vertex multiplication and examine the load-balancing with respect to different implementations of communication required by the multiplication.

4.1. Formulation

Parallel implementation of the coarse space vertex degrees of freedom solver requires partitioning the global domain into non-overlapping groups of elements. To minimize the volume of communication between different groups it is preferable to form a partition of one or few clusters of adjacent elements; in our code this task is performed by Metis. Given such compact partitioning of elements, we can identify another boundary–interior decomposition with respect to the vertex degrees of freedoms. This decomposition is highlighted in Fig. 6 where we consider an illustrative example of a partitioning of a 2D triangular computational domain into four partitions; an analogous extension to 3D domain is reasonably straightforward. In this figure the interfaces between partitions are identified by wide lines. Vertices shared by adjacent elements are sub-divided into two groups: (a) boundary–boundary vertices (shown as squares), located on the interfaces between partitions, and (b) interior–interior vertices (shown as circles), located inside each partition.

Our aim is to precondition the global Schur complement system \( \mathbf{S} \). We can order \( \mathbf{S} \) by vertex, edge and face degrees of freedom so as to obtain a matrix structure of the form

\[
\mathbf{S} = \begin{bmatrix}
S_{vv} & S_{ve} & S_{vf} \\
S_{ev}^T & S_{ee} & S_{ef} \\
S_{fv}^T & S_{ef}^T & S_{ff}
\end{bmatrix},
\]

where the subscripts \( v, e \) and \( f \) refer to vertex, edge and face degrees of freedom, respectively. As described in Section 3.1, the \( p \)-type (polynomial) scaling of the system is preconditioned through the numerical change of basis associated with the low energy basis (block diagonal) preconditioner. To take account of the \( h \)-type (elemental) scaling in our preconditioning strategy an appropriate solution is to invert the vertex space sub-matrix of the Schur complement, i.e., \( S_{vv} \). As we mentioned in Section 3.1, we use an additive Schwartz preconditioner, which includes two parts: the coarse space linear vertex block and the low energy preconditioner constructed by the numerical orthogonalization of the basis [6].
\[
\begin{bmatrix}
S_{vv}^{-1}
\end{bmatrix}
+ \begin{bmatrix}
\begin{bmatrix}
\operatorname{Diag}[S]\end{bmatrix}
S_{eb}
0
\end{bmatrix}
\begin{bmatrix}
R_f
S_{eb}^{-1}
0
\end{bmatrix}
R_v,
\]
where \( \operatorname{Diag}[S] \) is the diagonal of \( S \) vertex modes, \( (S_{eb})_{eb} \) is the block diagonal of the edge (face) components \([6]\). Note that the Schur complement \( S \) has already been orthogonalized with respect to the bubble modes of the 3D expansion and in this sense might be considered as having low energy within the interior of an element. However, the vertices will not have low energy within the “wire-basket” space of the tessellated elements.

We can now further decompose the coarse space linear vertex system \( S_{vv} \) into degrees of freedom on the boundary of the partitions (denoted again with a subscript \( b \)) and those degrees of freedom within the partitions (denoted with a subscript \( i \)) to obtain
\[
S_{vv} = \begin{bmatrix}
V_{bb} & V_{bi} \\
V_{ib} & V_{ii}
\end{bmatrix}
\begin{bmatrix}
\hat{v}_b \\
\hat{v}_i
\end{bmatrix}
= \begin{bmatrix}
g_b \\
g_i
\end{bmatrix},
\]
where we have used \( \hat{v} \) and \( g \) to denote the solution and the forcing vectors corresponding to the vertex components; the \( V_{bb} \) sub-matrix corresponds to the boundary–boundary modes; \( V_{ii} \) correspond to the interior–interior modes, and \( V_{ib} = V_{bi} \) is the sub-matrix which couples these two systems. Analogously, \( \hat{v}_b \) and \( \hat{v}_i \) are the solution coefficients of the boundary and the interior degrees of freedom, marked in illustration of Fig. 6 by squares and circles, respectively. Once again, we can apply substructuring to decouple the interior and boundary degrees of freedom, i.e.,
\[
\begin{bmatrix}
V_{bb} - V_{bi}[V_{ii}]^{-1}V_{ib} & 0 \\
V_{ib} & V_{ii}
\end{bmatrix}
\begin{bmatrix}
\hat{v}_b \\
\hat{v}_i
\end{bmatrix}
= \begin{bmatrix}
g_b - V_{bi}[V_{ii}]^{-1}g_i \\
g_i
\end{bmatrix}.
\]
The matrix \( V_{bb} - V_{bi}[V_{ii}]^{-1}V_{ib} \) is the Schur complement matrix of \( S_{vv} \) based on degrees of freedom along the partition. Further, by construction \( V_{ii} \) has a block-diagonal structure that can be readily inverted, where each block has a size corresponding to the number of interior vertices on a given partition. For a fixed mesh size, the number of boundary–boundary degrees of freedom depends on the number of partitions, and therefore the more partitions that are generated the larger the size of \( V_{bb} \) and the number of blocks in \( V_{ii} \). In contrast, the size of \( V_{ib} \) decreases as the number of partitions grows. Solution for \( \hat{v}_i \) can be computed locally within each partition once the values of \( \hat{v}_b \) have been determined.

The solution for \( \hat{v}_b \) can be computed locally by constructing the global system on each processor or using a parallel approach. For the relatively small amount of elements one typically uses in a spectral/hp element discretization as compared to a classical finite element methods such a local approach has been successfully applied for up to a few tens of processors. However, this approach will clearly saturate when considering larger number of processors and the number of elements typically required by large-scale computation. In the following sections, we discuss alternative approaches for solving the boundary–boundary vertex system, and then provide details on assembling the boundary–boundary system \( V_{sc} = V_{bb} - V_{bi}[V_{ii}]^{-1}V_{ib} \) in parallel.

4.2. Algorithms for solution of boundary–boundary system

We considered two approaches for solving the boundary–boundary system
\[
[V_{sc}]\hat{v}_b = g_b - V_{bi}[V_{ii}]^{-1}g_i.
\]
The first approach is based on the LU decomposition of the operator \( V_{sc} \), and the second is directly inverting the \( V_{sc} \) operator. Each of the two methods can be executed in serial or in parallel. In the serial approach the LU decomposition of operator \( V_{sc} \) (or its inverse) is replicated on each processor, while in the parallel one it is distributed over all processors.

As we mentioned before, when the number of partitions is small and consequently the rank of \( V_{sc} \) is low, it might not be beneficial to parallelize the solution of the boundary–boundary system. We implemented the serial and the parallel approaches for solving the system by inverting the \( V_{sc} \) operator (in the preprocessing only) and performing matrix-vector multiplication at every iteration; in Fig. 7 we compare the overall parallel efficiency of our solver. The rank of \( V_{sc} \) operator for the Poisson and Helmholtz solvers is presented in Table 2. Clearly, the serial solver for system \((8)\) is not scalable on more than 64 processors.

Next we compare the performance of our solver where the system \((8)\) is solved in parallel by means of the LU decomposition or with the inversion of the operator \( V_{sc} \); we used functions \texttt{pdgetrs} and \texttt{pdgemv} from ScaLapack library \([12]\) for the parallel LU solve and for the matrix-vector multiplication, respectively. In both cases, the LU decomposition and inversion of \( V_{sc} \) was done during preprocessing and the time of the preprocessing was excluded; we plot the results in Fig. 8. The poor efficiency of the LU based solver is due to high volume of communication together with relatively high number of floating point operations. ScaLapack does not implement sparse algebra operators, and the LU decomposition is stored as a full dense matrix in 2D block cyclic format, hence the high number of floating point operations.

In preliminary tests we also implemented the parallel version of SuperLU library (version 2.0) \([13]\) for solution of system \((8)\). SuperLU uses a sparse algebra and is efficient for solution of linear systems with very sparse operators. Although the
number of non-zero values (nz) of $V_{SC}$ is small (see Fig. 9), they are not packed in a way which will maximize the efficiency of memory access to the values of LU. Compared to ScaLapack-based solver, the SuperLU-based solver, applied to (8), performed about five times slower on CRAY XT3 and showed poor scalability.
Based on the performed tests we concluded that the most efficient way to solve the system (8) is to directly invert the operator and to perform a parallel matrix-vector multiplication. Our next tasks are to develop a technique to assemble the VSC in parallel while minimizing memory requirements and find the most efficient procedure for the parallel matrix-vector multiplication. In the following section, we concentrate on the construction and inversion of VSC.

4.3. Construction of coarse space linear vertex operator

The parallel construction of the coarse space linear vertex operator is performed during the preprocessing stage in three steps:

1. Values of VSC corresponding to each partition are computed locally.
2. Processors perform a ring-type communication to redistribute locally computed values of VSC.
3. LU decomposition and inversion of VSC is performed in parallel with subsequent redistribution from 2D block cyclic format to standard block decomposition.

In the following we discuss these steps in detail.

The first step of VSC construction consists of computing the local contribution, namely $V_{SC}^{k}(i,j)$, where $i$ and $j$ are global indices and $k$ is the partition ID. At this point, our assumption is that the values $V_{SC}(i,j)$ may have random distribution across processors and any amount of overlap is allowed. Indeed, the distribution of $V_{SC}(i,j)$ depends on the global numbering of the boundary–boundary degrees of freedom and on the partitioning of the computational domain. In Fig. 10 we sketch distribution of values of $V_{SC}(i,j)$ computed for the velocity system, in the computational domain of carotid artery (see Fig. 7(right)) partitioned into four parts. To minimize the memory requirement, values of $V_{SC}^{k}(i,j)$ are stored in sparse matrix using the coordinate format [14]; moreover, the upper triangular part of symmetric operator VSC is not stored. The ordering of $V_{SC}^{k}(i,j)$ stored in sparse matrix format is not important. Each vertex can be shared by several elements within partition. If a vertex contribution with the same $i$, $j$ indices is already stored we add the value of $V_{SC}^{k}(i,j)$ to the existing one, otherwise we increase the size of the matrix and append a new value with corresponding indices. It is not necessary to compute the exact number of values stored in the sparse matrix prior to its construction, since we can use dynamic memory reallocation to store new values. We note that the dynamic memory reallocation for the matrix is an expensive computation task; one way to minimize the computational cost of resizing the matrix is to allocate some extra memory in advance. However, the discussed procedure is performed only in the preprocessing stage and typically requires less than one second.

In the second step in the parallel construction of VSC is to redistribute the locally computed values between partitions. In NEKTAR we implement two matrix partitioning methods. In Fig. 11 we present decompositions of a matrix $V$ onto six partitions. The left plot shows the standard block decomposition. The right plot depicts the corresponding two-dimensional block cyclic distribution. Each block $V_{rs}$ is a square matrix of rank $N_{rs}$, $r = [1N_r]$ and $c = [1N_c]$ - are the block row and column index. If mod(rank(VSC), $N_{rs}) = 0$ then the blocks $V_{rs}$ are not square matrices; although, from algorithmic point of view this is perfectly acceptable, it alters the computational load balance. Scalapack assumes matrices are laid out in a two-dimensional block cyclic decomposition. We use a basic block of size $N_{rs} = 4, \ldots, 8$. This size is not very efficient for
LU decomposition and subsequent matrix inversion (performed only during preprocessing) where the optimal size of a block is between 30 and 80. However, the small size of the basic block leads to better load balancing in parallel matrix-vector multiplication, performed (by ScaLapack) at each iteration. As for the 2D processor grid layout, our numerical experiments indicate that better performance is achieved when the number of processors-columns is greater or equal to the number of processors-rows.

Given the alignment of processors where \( V_{SC} \) is to be distributed, the size of the base block and considering the rank of \( V_{SC} \), we allocate only the necessary memory for sub-matrix of \( V_{SC} \) on each processor. To map \( V_{SC} \) into two-dimensional blocks with cyclic distribution required by ScaLapack, sparse matrices with values and indices of \( V_{SC}(i,j) \) are circulated between processors (employing ring-type communication) where the local part of two-dimensional block cyclic matrix \( V_{SC2D} \) is updated with relevant values from \( V_{SC}(i,j) \).

\[ \text{CPU 0} \quad \text{CPU 1} \quad \text{CPU 2} \quad \text{CPU 3} \]

\[ \text{nz} = 156 \quad \text{nz} = 169 \quad \text{nz} = 325 \quad \text{nz} = 990 \]

**Fig. 10.** Initial distribution of \( V_{SC}(i,j) \) computed for the velocity system. Computational domain of carotid artery (see Fig. 7(right)) is sub-divided into four partitions. Due to the symmetry of the operator only low triangular terms are stored; \( \text{nz} \) – number of non-zero components.

**Fig. 11.** Parallel construction of \( V_{SC} \). Matrix is distributed onto six partitions on \( 3 \times 2 \) two-dimensional processor grid. Wide lines represent decomposition of the matrix to six partitions. \( V_{SC} \) – block of rank \( N_{BS} \), \( r \) and \( c \) are the row and the column index of each block. Left: standard decomposition; right: two-dimensional block cyclic decomposition used in ScaLapack. Colors represent standard blocks (sub-matrices of a rank \( N_{BS} \)), which under 2D block cyclic mapping are grouped on the same processors.
At the third step we use ScaLapack for parallel LU decomposition and then parallel matrix inversion. As we already mentioned, the size of a basic block in 2D cyclic distributed matrix is not optimized for parallel LU decomposition and matrix inversion, however these two operations require only 0.5–5 s. After the parallel matrix inversion is performed, we can use the $\left(V_{SCD}\right)^{-1}$ distributed over all processors in 2D block cyclic format for parallel matrix-vector multiplication to compute $v_b = \Pi^{-1}(V_{SCD})^{-1}\Pi(g_b - V_{bi}[V_{ii}]^{-1}g_i)$ using the ScaLapack pdgemv function. The operator $\Pi$ is permutation operator which maps a vector into block cyclic format. Alternatively, we can map $(V_{SCD})^{-1}$ to form $(V_S)^{-1}$, which is distributed in standard blocks over all processors, as illustrated in Fig. 11. We found that the latter approach is advantageous due to additional optimization we can apply for redistribution vectors $v_b$ and $(g_b - V_{bi}[V_{ii}]^{-1}g_i)$ over processors. We provide details of gathering and scattering of these vectors in Section 4.5.

4.4. Load balancing in parallel matrix vector multiplication

The efficiency of parallel matrix-vector multiplication depends strongly on two factors: (a) effectiveness of communication between processors, and (b) serial performance of matrix vector multiplication. In the original ScaLapack pdgemv function and modified for NEKTAR pdgemv function the serial part of matrix-vector product is based on the same BLAS library. However, the communication patterns are different. In Fig. 12 we compare communication balance during the matrix vector multiplication performed by ScaLapack pdgemv function and by the NEKTAR pdgemv function. In this simulation 128 processors laid out in $8 \times 16$ process grid were used. The first 16 processors aligned on the row zero, the next 16 (processor IDs 16 to 31) on the first row, and so on. Processors ID on the zero column of the 2D process grid can be computed from

![Fig. 12. Blue Gene: load imbalance in preconditioning stage. Upper plot – simulation with ScaLapack pdgemv function. Lower plot – simulation with NEKTAR pdgemv function. Problem size: 19,270 tetrahedral elements, fifth-order polynomial approximation. Simulation performed on IBM Blue Gene supercomputer using 128 CPUs. Performance was monitoring with IPM tool [15]. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.]

ID_{row,col=0} = row \times 16, \text{row} = 0, \ldots, 7. As shown in Fig. 12 (upper plot), a severe load balance problem occurs when MPI_Bcast is called. Additionally, the processors from column zero spend more time in MPI_Allreduce than those from other columns. This situation is due to two factors: (a) ScaLapack pdgemv function expects that the vector, which multiplies the distributed matrix, is supplied to the column zero of process grid only in cyclic block-row distribution, then it is redistributed among other processors in cyclic block-column distribution and (b) ScaLapack pdgemv function returns the result in cyclic row format to the processors of the column zero only. Thus, the result should be reordered and then broadcast from the column zero to the rest. While the result of matrix-vector multiplication is reordered on the column zero (this operation also includes one communication between processors of column zero), processors on other columns are waiting.

By modifying the ScaLapack pdgemv function we could eliminate unnecessary data transfer from one processor to another and optimize redistribution of multiplication result to all processors. As a result we, achieved better load balancing and also reduced the total execution time by 10–30% as presented in Table 3.

4.5. Implementation of parallel matrix-vector multiplication in NEKTAR

In Fig. 13 we sketch the distributed matrix \((V_{sc})^{-1}\). For illustration purposes, the matrix is distributed over a six-processor mesh, aligned in two rows and three columns. In the following for simplicity of notations we denote by \(x\) and \(y\) the solution vector and forcing term.

We consider the parallel matrix-vector multiplication as a three-step procedure:

- **step 1** – parallel assembling of the vector \(y\);
- **step 2** – local sub-matrix-sub-vector multiplication;
- **step 3** – redistributing the results of the last operation between partitions.

On **step 1**, local contribution to vector \(y\) is computed on each processor and then redistributed. We note that we use the spectral element code where domain decomposition is done element-wise by Metis [10], while the dense matrix \((V_{sc})^{-1}\) is distributed regardless of the element-wise domain decomposition. The latter requires to develop an efficient and balanced procedure for parallel construction of the vector \(y\) and later for distribution of the result of matrix-vector product stored in \(x\).

For communication between processors we split the default MPI communicator global MPI_COMM_WORLD into MPI_COMM_ROW and MPI_COMM_COLUMN sub-communicators in order to perform message passing within the scope of each row or column of the process grid.

Redistribution of the vector \(y\) is done in three stages:

a) On each processor we compute the local contribution to the vector \(y\).

b) On every row of the processor mesh we distribute values of \(y\) such that on processors of each column we compute \(\sum_{k=0}^{N_{row}} y_{jk}\), where \(N_{col}\) is number of columns in 2D grid of processors, \(j\) is the global column index in the range \(J_s <= j <= J_e\). Here \(J_s\) and \(J_e\) are the global index of the first and the last column of the local block of \((V_{sc})^{-1}\).

\[
\text{Fig. 13. Parallel matrix-vector multiplication } (V_{sc})^{-1}y = x. \text{ The operator } V_{sc} \text{ is distributed on } 3 \times 2 \text{ processors grid. The vector } y \text{ is partitioned into three sub-vectors according to number of columns, and vector } x \text{ is partitioned into two sub-vectors according to number of row of the processor grid.}
\]
c) Finally, values of $y_j, j = j_1 <= j <= j_2$ are summed on each column of the processor mesh using MPI_Allreduce and MPI_COMM_COLUMN communicator.

In Fig. 14 we provide an illustration of the elements redistribution within each row of processors. In the illustration the vector $y$ has a length of 10; on the processors of column zero values of $y_j, j = 0, \ldots, 3$ are required, on the processors of column one values $y_j, j = 4, \ldots, 7$ are required, and on the processors of the last column values of $y_j, j = 8, 9$ are required for matrix-vector multiplication. The shaded fields correspond to values of $y_j$ computed in each partition according to element-wise domain decomposition.

We considered several MPI implementations to perform step 1. In the following we provide the description of four different methods we implemented. For clarity we provide a pseudocode for these four methods in Appendix C.

Our first version (V1) uses MPI_Allreduce within MPI_COMM_WORLD for a global reduction operation over the entire length of the vector $y$. These operations transmit more data than is strictly necessary, since each processor only needs to get data for $y_j$ with $j$ bounded by $j_1 <= j <= j_2$. For the example of Fig. 14 the length of message will be 10. In addition, the number of arithmetic operations in this case is higher, since all values of vector $y$ are passed and summed on each processor.

In the second version (V2) we employ the aforementioned sub-communicators MPI_COMM_ROW and MPI_COMM_COLUMN. First, MPI_Allreduce operation is done within each row to perform MPI_SUM of the vector $y$ over its entire length; then MPI_Allreduce is done within each column. This version communicates significantly less data than V1, yet there are still some redundant data (zeros) being transmitted in a scope of a row.

In the third version (V3) we employ MPI_Alltoallv instead of MPI_Allreduce in rows, while the second MPI_Allreduce is the same as in V2. The MPI_Alltoallv is a blocking function, however, performed over the row communicator-MPI_COMM_ROW it synchronizes processor within a scope of each row only. This version does not communicate unnecessary data in a scope of a row.

In the fourth version (V4) we replace the blocking call MPI_Alltoallv with point-to-point, non-blocking sends and receives (MPI_Isend and MPI_Irecv) operations, followed by MPI_Waitany; while the MPI_Allreduce operation within columns is the same as in V2.

These four versions were studied in detail on the three architectures we had access to (IBM Blue Gene, Cray XT3 and IBM Power 4). We found that on sufficiently large partitions of Blue Gene (over 2048 CPUs) versions V1 and V3 perform with almost the same speed and are better than the other two versions. The explanation of this behavior has several components:

1. Global MPI_Allreduce is done over the tree interconnect of Blue Gene, which is specialized for collective communication, while collective calls over subcommunicators are done using the 3D torus interconnect.
2. There is only one MPI call with associated latency instead of two calls as in V2 and several calls in the V4.
3. The reason that V4 version does not perform as well as V3 most likely has to do with the fact that vendor-implemented MPI_Alltoallv optimizes network traffic to avoid hotspots.

On IBM Power4 (Datastar) the cost of global MPI_Allreduce was more than twice higher than row-wise and column-wise MPI_Allreduce. For example, using 512 processors of IBM Power 4 construction of the vector $y$ with a global MPI_Allreduce (V1) call was accomplished in 0.093 seconds, while using the alternative approach (V2) 0.040 seconds where required. This behavior is more intuitively understandable since in the latter approach less data are sent. On CRAY XT3 we likewise saw a behavior different than the one on Blue Gene: V4 performed faster than V1, V2 and V3. This demonstrates the different characteristics of interconnects of different platforms and the necessity to target algorithm development correspondingly. In the benchmark study presented in the following section we focus on the best algorithm on a given platform.

The step 2 of parallel matrix-vector multiplication requires only one call to level 2 BLAS function only and no communication is performed. On each processor we multiply the local part of $(V_{SC})^{-1}$ by the local part of $y$ to obtain local part of $x$.

On the step 3, to redistribute the result of the parallel matrix vector product we implement the following procedure: first, we sum the result in each row of the processors concurrently using MPI_Allreduce and MPI_COMM_ROW communicator. The size of the message here is equal to the number of $(V_{SC})^{-1}$ rows stored on each row of the process grid. Second, within the scope of each column we implement non-blocking communication to exchange values of $x$ between processors, the algorithm is similar to V4 used for gathering values of the vector $y$. We note that the processor in row $i$ receives only the required values of $x$ from other processors in its column, according to the original element-wise domain decomposition.
5. Parallel performance results

In this section we investigate the performance of low energy basis and diagonal preconditioner used in NEKTAR. We compare the scalability and effectiveness of the two preconditioners. The Schur complement of the Mass and the Stiffness operators constructed with the polynomial basis used in NEKTAR are diagonally dominant, and the values on their diagonals are varying, and this makes the diagonal preconditioning quite effective, unlike in other methods. In Table 4 we compare the number of iterations required for the solution of Helmholtz equation $\nabla^2 U - \lambda U = -\left(\lambda + 6\pi^2\right)\sin(\pi x)\cos(\pi y)\cos(2\pi z)$ with Conjugate Gradient solver and three types of preconditioners:

(a) The diagonal preconditioner, defined as the inverse of the main diagonal of the Schur complement $S$.

(b) The Block Preconditioner, defined as

\[
\begin{bmatrix}
S_{vv}^{-1} & S_{ve}^{-1} \\
S_{ee}^{-1} & S_{ff}^{-1}
\end{bmatrix},
\]

where $S_{ve}^{-1}$ and $S_{ff}^{-1}$ are constructed by inverting the blocks of $S_{ee}$ and $S_{ff}$ corresponding to coupling of the edge (face) $i$ with itself only, i.e., omitting the coupling of the edge (face) $i$ with the edge (face) $j$.

(c) the Coarse Space Linear Vertex Block Preconditioner, defined as

\[
\begin{bmatrix}
S_{vv}^{-1} \\
I
\end{bmatrix}
\]

For the discretization we use 96 tetrahedral elements, and the iterations are terminated when the residual is less than $10^{-6}$. We observe that significant reduction in the iteration count is achieved with the simple diagonal preconditioner.

Results presented in this section show that the computational cost of simulations of unsteady flows is considerably lower with LEBP than with the standard diagonal preconditioner. Our results also show that scalability of the more effective LEBP is practically the same as of diagonal preconditioner. In all our tests we terminate the conjugate gradient iterations when the relative error in the residual is less than $10^{-8}$.

Simulations of unsteady flow typically require integration over $10^5$-$10^6$ time steps and at each time step 20 to 100 iterations are executed on average. Thus, the cost of solution of preconditioning outweighs significantly the cost of construction of the LEBP, hence it is not included in the performance study.

5.1. Performance of low energy basis and diagonal preconditioner

First we compare the iteration count using the diagonal preconditioner and LEBP. Our computational domain has a shape of cylindrical pipe and is constructed of 67,456 tetrahedral elements; the initial condition for velocity fields is $v = 0$. At the inlet of the domain we prescribe a Poiseuille velocity profile, while at the outlet we use fully developed flow is assumed. We monitor the number of iterations during the first 1000 time steps; we observe (see Fig. 15) that the number of iterations at the beginning of simulation is higher than after some transient period, thus the performance of numerical solver with the two preconditioners is compared after a few hundreds of time steps.

In Fig. 16 and Table 5 we compare the mean cpu-time per time step required for 3D flow simulation with LEBP and with Diagonal Preconditioner; almost an eightfold speed-up in the execution time is observed. In this simulation the solution was integrated 1000 time-steps and the mean cpu-time per time step was measured as an average of the cpu-time required for the last 10 time steps of simulation. In the last column of Table 5 we present the Rank of $V_{sc}$. Note that with increasing number of processors the size of the local block of $V_{sc}$ is decreased, which means that the computation versus communication ratio is decreasing as well.

Performance of NEKTAR with LEBP was also tested on CRAY XT3 computer of ERDC, which has dual-core AMD Opterons 2.6 GHz processors and 2 GB memory per core (but only 1.8 GB is available for applications). Achieving good scalability on a cluster of processors with relatively high clock-speed requires very good optimization of communication algorithms since the time spent on computation is usually 2–4 times less than on computers with relatively slow processors such as Blue

<table>
<thead>
<tr>
<th>$P$</th>
<th>No preconditioners</th>
<th>Diagonal preconditioner</th>
<th>Block preconditioner</th>
<th>Vertex block preconditioner</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>162</td>
<td>36</td>
<td>36</td>
<td>125</td>
</tr>
<tr>
<td>6</td>
<td>400</td>
<td>71</td>
<td>67</td>
<td>312</td>
</tr>
</tbody>
</table>
Gene. In Fig. 17 and Table 6 we show the scalability of NEKTAR achieved on the CRAY XT3 computer. The data presented in Fig. 17 are based on a simulation in a domain of 120,813 tetrahedral elements with 6th, 8th and 10th polynomial approx-

![Fig. 15. Number of PCG iterations for three velocity components and pressure. Upper plots: number of iterations: solid line – LEBP, dash line – diagonal preconditioner. Lower plots – reduction in iteration count. Problem size: 67,456 tetrahedral elements with eight-order polynomial approximation.](image)

![Fig. 16. Blue Gene (SDSC): mean cpu-time for simulation with Diagonal and low energy basis preconditioner. Problem size: 67,456 tetrahedral elements with eight-order polynomial approximation. Computation performed using 1 processors per node.](image)

<table>
<thead>
<tr>
<th>No. of CPUs</th>
<th>Diagonal preconditioner (s)</th>
<th>Low energy preconditioner (s)</th>
<th>Rank of $\text{VSC}$: velocity (pressure)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>24.1</td>
<td>2.88</td>
<td>5449 (8596)</td>
</tr>
<tr>
<td>1024</td>
<td>12.43</td>
<td>1.61</td>
<td>6625 (10,577)</td>
</tr>
<tr>
<td>2048</td>
<td>6.69</td>
<td>0.98</td>
<td>7552 (12,192)</td>
</tr>
<tr>
<td>3072</td>
<td>N/A</td>
<td>0.76</td>
<td>8018 (13,072)</td>
</tr>
</tbody>
</table>

Gene. In Fig. 17 and Table 6 we show the scalability of NEKTAR achieved on the CRAY XT3 computer. The data presented in Fig. 17 are based on a simulation in a domain of 120,813 tetrahedral elements with 6th, 8th and 10th polynomial approx-
Fig. 17. CRAY XT3 (ERDC): left: performance of NEKTAR with LEBP. Parallel speed-up. Problem size: 120,813 tetrahedral elements with 6th, 8th and 10th order polynomial approximation. Right: geometry of the computational domain, color corresponds to pressure values.

Table 6
CRAY XT3 (ERDC): mean cpu-time for simulation with LEBP. Problem size: 120,813 tetrahedral elements with 6th, 8th and 10th order polynomial approximation. $N_{\text{points}}$ is a total number of quadrature points for each unknown.

<table>
<thead>
<tr>
<th>Number of CPUs</th>
<th>$P = 6$</th>
<th>$P = 8$</th>
<th>$P = 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.64 s</td>
<td>1.3 s</td>
<td>2.5 s</td>
</tr>
<tr>
<td>1024</td>
<td>0.39 s</td>
<td>0.74 s</td>
<td>1.37 s</td>
</tr>
<tr>
<td>2048</td>
<td>0.28 s</td>
<td>0.46 s</td>
<td>0.78 s</td>
</tr>
<tr>
<td>4096 (*)</td>
<td>0.50 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$N_{\text{points}}$</td>
<td>69,588,288</td>
<td>132,894,300</td>
<td>226,161,936</td>
</tr>
</tbody>
</table>

* The result for 4096 CPUs was obtained on CRAY XT3 of the Pittsburgh Super Computing Center; in our experience this computer is typically 5–10% slower than CRAY XT3 of ERDC.

Fig. 18. CRAY XT3 (PSC): performance of NEKTAR with LEBP. Cpu-time balance. Problem size: 19,270 tetrahedral elements with 4th (left) and 5th (right) order polynomial approximation. Computational domain of carotid artery, illustrated in Fig. 7. Computation performed on 128 CPUs.
imation. In each simulation the solution was integrated for 500 time steps and the mean cpu-time was computed based on the last 10 time steps.

In Fig. 18 we plot the cpu-time balance monitored during the first 40 iterations required by the Poisson solver in simulation of unsteady flow in a domain carotid artery (Fig. 7). For this simulation we used 128 processors of CRAY XT3. The plots show that about half of the cpu-time is consumed by preconditioning. Only 15% of the cpu-time consumed by preconditioning is spent on solving system [8]; this time includes the three steps of the parallel matrix-vector multiplication as discussed in Section 4.5. The diagonal preconditioning requires only one vector–vector multiplication and no communication; thus the time spent on the preconditioning is negligible, and the total cpu-time required for one iteration in the last simulation would be of order $4E-3$ to $5E-3$ s.

6. Summary

We developed a scalable and efficient preconditioner for solution of elliptic equations using the spectral/hp element method. The preconditioner was successfully applied to simulations of unsteady bioflows in domains with complex geometry, where the elliptic solves is the dominant cost. The number of iterations was reduced by factor 10–50 while the overall computational cost of the simulation was reduced by an order of magnitude when a standard diagonal preconditioner was substituted by the more efficient low energy basis preconditioner. Platform-specific optimizations were considered in the implementation and simulations were performed on the IBM Blue Gene and on the CRAY XT3. Very good scalability in the solution of large scale problems up to 4096 processors (available for the current work) was achieved. In particular the parallel efficiency increases with the polynomial order of the spectral/hp element.

Acknowledgements

This work was supported by the National Science Foundation’s CI-TEAM program under Grant OCI-0636336. Supercomputer resources were provided by the NSF Large Allocations Resource Committee at San Diego Supercomputer Center, DOD’s ERDC and Pittsburgh Supercomputing Center. We want to thank the San Diego Supercomputer Center for their support of this project through the Strategic Applications Collaboration program. We also like to thank Dr. David O’Neal of PSC for his support. We want to acknowledge the reviewers for provided comments and suggestions which helped to improve our paper.

Appendix A. Numerical scheme in NEKTAR

Numerical solution of the Navier–Stokes Eq. (1)

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} = -\nabla p + \nu \nabla^2 \mathbf{v}^{n+1}, \quad \nabla \cdot \mathbf{v} = 0$$  \hspace{1cm} (A.1)

in NEKTAR is performed using high-order time splitting scheme [8], that decouples the velocity and pressure fields, i.e.,

$$\mathbf{v}^* = \sum_{k=0}^{J^e-1} \alpha_k \mathbf{v}^{n-k} - \Delta t \left( \sum_{k=0}^{J^e-1} \beta_k (n \mathbf{l})^{n-k} + \mathbf{f} \right), \quad \mathbf{n} \mathbf{l} = \nabla \cdot (\nabla \mathbf{v}),$$  \hspace{1cm} (A.2a)

$$\nabla^2 p = \frac{1}{\Delta t} \nabla \cdot \mathbf{v}^*,$$  \hspace{1cm} (A.2b)

$$\gamma_0 \mathbf{v}^{n+1} = \mathbf{v}^* + \Delta t (-\nabla p + \nabla^2 \mathbf{v}^{n+1}),$$  \hspace{1cm} (A.2c)

where $J^e$ is the time splitting order, $\gamma_0$ and $\alpha_k$ are coefficients of backward differentiation formula and $\beta_k$ are coefficients of Adams–Bashforth integration scheme.

In space spectral/hp-element method discretization for the velocity and pressure fields is applied. The computational domain used by NEKTAR consists of polymorphic elements, e.g., tetrahedra, hexahedra, prisms, pyramids or a combination of these. Within each element the solution is approximated in terms of hierarchical, mixed-order, semi-orthogonal Jacobi polynomial expansions [7]. They are hierarchical in a sense that the modes are separated into vertex (linear term) $\Phi_k$, edge $\Psi_k$, face $\Theta_k$ and interior or bubble modes $A_k$. The polynomial approximation of a field $V(t, \mathbf{x})$ at any point $\mathbf{x}$ is given by

$$V(t, \mathbf{x}) = \sum_{k=1}^{N_v} V_k(t) \Phi_k(\mathbf{x}) + \sum_{k=1}^{N_e} V_k^e(t) \Psi_k(\mathbf{x}) + \sum_{k=1}^{N_f} V_k^f(t) \Theta_k(\mathbf{x}) + \sum_{k=1}^{N_i} V_k^i(t) A_k(\mathbf{x}),$$  \hspace{1cm} (A.3)

where $N_v$ is the number of vertex modes and $N_e = (P-1)(\# \text{ of edges})$, $N_f = (P-2)(\# \text{ of faces})$ and $N_i = (P-3)$ are the number of the edge, face and interior modes, respectively, and $P$ is polynomial order of the expansion; the $N_l$ term represents the truncation error.
The weak formulation of the Eqs. (2b) and (2c) is obtained by Galerkin projection, namely

$$\mathbf{L}\hat{\phi} = -\frac{1}{\Delta t}(\nabla \cdot \mathbf{v}, \phi) + \left( \frac{\partial p}{\partial n}, \phi \right)$$  \hspace{1cm} (A.4a)

and

$$\mathbf{H}\hat{\mathbf{v}}^{n+1} = \frac{1}{\Delta t_0}(\mathbf{v} - \Delta t \nabla p, \phi) + \frac{\Delta t}{\Delta t_0} \left( \frac{\partial \mathbf{v}}{\partial x}, \phi \right)^{n+1},$$  \hspace{1cm} (A.4b)

where $\mathbf{H} = \mathbf{M} - \frac{\Delta t}{\Delta t_0} \mathbf{L}$, and $\mathbf{M}$ and $\mathbf{L}$ are the mass and stiffness matrices, respectively. The quantities with hat denote modal amplitudes, as the linear solves are performed in the modal domain. The Neumann pressure boundary condition at the boundaries with prescribed velocity are computed from

$$\frac{\partial p}{\partial n} = \sum_{k=0}^{\text{Je}-1} \left[ \left( -\frac{\partial \mathbf{v}}{\partial t} - \mathbf{n} \mathbf{v} \times (\nabla \times \mathbf{v}) \right) \cdot \mathbf{n} \right]^{n-k}.$$  \hspace{1cm} (A.5)

Appendix B. Construction of the elemental transformation matrix $\mathbf{R}$

We recall that

$$S_2 = \mathbf{RS}_1 \mathbf{R}^T$$

and consider a single elemental matrix and the transformation of basis which arises from a matrix $\mathbf{R}$ of the form:

$$\mathbf{R} = \begin{bmatrix} 1 & \mathbf{R}_{re} & \mathbf{R}_{rf} \\ 0 & 1 & \mathbf{R}_{ef} \\ 0 & 0 & 1 \end{bmatrix},$$

where the vertex modes are listed first followed by the edge and face modes. The matrices $\mathbf{R}_{re}, \mathbf{R}_{rf}$ represent the modification of the vertex modes by the edges and face modes. Similarly the matrix $\mathbf{R}_{ef}$ represents the modification of the edge modes by the face modes. The transformation matrix has upper triangular structure and hence it is easily invertible.

Let us define $\mathbf{R}$ as

$$\mathbf{R} = \begin{bmatrix} 1 & \mathbf{R}_v \\ 0 & \mathbf{A} \end{bmatrix} \text{ where } \mathbf{R}_v = [\mathbf{R}_{re}, \mathbf{R}_{rf}], \quad \mathbf{A} = \begin{bmatrix} 1 & \mathbf{R}_{ef} \\ 0 & 1 \end{bmatrix}$$

and write the original Schur complement of the Helmholtz matrix as

$$\mathbf{S}_1 = \begin{bmatrix} \mathbf{S}_{vv} & \mathbf{S}_{ve} & \mathbf{S}_{vf} \\ \mathbf{S}_{ev} & \mathbf{S}_{ee} & \mathbf{S}_{ef} \\ \mathbf{S}_{fv} & \mathbf{S}_{fe} & \mathbf{S}_{ff} \end{bmatrix} = \begin{bmatrix} \mathbf{S}_{vv} & \mathbf{S}_{ve} & \mathbf{S}_{vf} \\ \mathbf{S}_{ev} & \mathbf{S}_{ee} & \mathbf{S}_{ef} \\ \mathbf{S}_{fv} & \mathbf{S}_{fe} & \mathbf{S}_{ff} \end{bmatrix}

\begin{bmatrix} \mathbf{S}_{ve} & \mathbf{S}_{ef} \\ \mathbf{S}_{ev} & \mathbf{S}_{ef} \end{bmatrix} \mathbf{A} \begin{bmatrix} \mathbf{S}_{ve} & \mathbf{S}_{ef} \\ \mathbf{S}_{ev} & \mathbf{S}_{ef} \end{bmatrix}^{-1},$$

then applying the transformation matrix, namely $\mathbf{S}_2 = \mathbf{RS}_1 \mathbf{R}^T$ we obtain

$$\mathbf{S}_2 = \begin{bmatrix} \mathbf{S}_{vv} + \mathbf{R}_{v} \mathbf{S}_{ve} + \mathbf{R}_{v} \mathbf{S}_{vf} \mathbf{R}_{T}^T \\ \mathbf{A} \mathbf{S}_{ve} + \mathbf{S}_{ef} \mathbf{R}_{T}^T \end{bmatrix}, \quad \mathbf{S}_2 = \mathbf{S}_{v} + \mathbf{R}_{v} \mathbf{S}_{ef} \mathbf{A}^T \begin{bmatrix} \mathbf{S}_{v} + \mathbf{R}_{v} \mathbf{S}_{ef} \mathbf{A}^T \\ \mathbf{A} \mathbf{S}_{v} + \mathbf{S}_{ef} \mathbf{R}_{T}^T \end{bmatrix}$$  \hspace{1cm} (B.1)

where

$$\begin{bmatrix} \mathbf{S}_{ve} + \mathbf{R}_{v} \mathbf{S}_{ef} + \mathbf{S}_{ef} \mathbf{R}_{T}^T \\ \mathbf{A} \mathbf{S}_{v} + \mathbf{S}_{ef} \mathbf{R}_{T}^T \end{bmatrix} = \begin{bmatrix} \mathbf{S}_{ve} + \mathbf{R}_{v} \mathbf{S}_{ef} + \mathbf{S}_{ef} \mathbf{R}_{T}^T \\ \mathbf{A} \mathbf{S}_{v} + \mathbf{S}_{ef} \mathbf{R}_{T}^T \end{bmatrix}$$  \hspace{1cm} (B.2)

In order to completely orthogonalise the vertex modes with the edge and face modes we require that

$$\mathbf{R}^T_v = -\mathbf{S}_{ef}^{-1} \mathbf{S}_{vef}^T.$$  \hspace{1cm} (B.3)

To decouple the edge modes from the face modes we see from inspecting (2) that

$$\mathbf{R}^T_{ef} = -\mathbf{S}_{ef}^{-1} \mathbf{S}_{ef}^T.$$  \hspace{1cm} (B.4)
Appendix C. Parallel matrix vector multiplication in NEKTAR

// z - vector of length Rank(V_SC) to store the forcing term and then
// the solution vector.
// Js - global index of first column of V_SC stored on this partition.
// Je - global index of last column of V_SC stored on this partition.
// tmp_buf - temporary buffer
/* compute x=[A]y */
gather_y_local(z, y_local, VERSION);
do_local_matrix_vector_mult(A_local, y_local, x_local);
scatter_x_local(x_local, z);

void gather_y_local(double *z, double *y_local, int VERSION){
  switch(VERSION){
    case V1:
      MPI_Allreduce(z,tmp_buf,length(z),MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
      memcpy(y_local,tmp_buf+Js,(Je-Js+1)*sizeof(double));
      break;
    case V2:
      MPI_Allreduce(z+Js,tmp_buf,Je-Js+1,
                    MPI_DOUBLE,MPI_SUM,MPI_COMM_ROW);
      MPI_Allreduce(tmp_buf,y_local,Je-Js+1,
                    MPI_DOUBLE,MPI_SUM,MPI_COMM_COLUMN);
      break;
    case V3:
      for (np=0; np < size(MPI_COMM_ROW); ++np){
        for (i=0, k=0; i < length(z); ++i){
          if (Js <= global_index_z[i] <= Je){
            if (z[i] != 0)
              sendbuf[np][k++]=z[i];
          }
        }
      }
      MPI_Alltoally(sendbuf,length_of_sendbufs,sdispls,MPI_DOUBLE,
                     recvbuf,length_of_recvbufs,rdispls,MPI_DOUBLE,
                     MPI_COMM_ROW);
      for (np=0; np < size(MPI_COMM_ROW); ++np)
        map_recvbuf_to_y_local(recvbuf[np],tmp_buf);
      MPI_Allreduce(tmp_buf,y_local,(Je-Js+1),
                    MPI_DOUBLE,MPI_SUM,MPI_COMM_COLUMN);
      break;
    case V4:
      MPI_Irecv(recvbuf[np],count[np],MPI_DOUBLE,np,tag,MPI_COMM_ROW,rqst_rcv[np]);
      for (np=0; np < size(MPI_COMM_ROW); ++np){
        for (i=0, k=0; i < length(z); ++i){
          if (Js <= global_index_z[i] <= Je){
            if (z[i] != 0)
              sendbuf[np][k++]=z[i];
          }
        }
      }
      MPI_Isend(sendbuf[np],length(sendbuf[np]),MPI_DOUBLE,
                np,tag,MPI_COMM_ROW,rqst_snd[np]);
      for (np=0; np < size(MPI_COMM_ROW); ++np){
        MPI_Waitany(np,rqst_rcv,&index,status);
        map_recvbuf_to_y_local(recvbuf[index],tmp_buf);
      }
      MPI_Allreduce(tmp_buf,y_local,(Je-Js+1),
                    MPI_DOUBLE,MPI_SUM,MPI_COMM_COLUMN);
  }
}
MPI_DOUBLE, MPI_SUM, MPI_COMM_COLUMN);
    break;
}  // end of switch

References